PRODUCT DOCUMENTATION

Pivotal™ Greenplum Database®

Version 6.11

Pivotal Greenplum Database Documentation

Rev: A03

Notice

Copyright

Revised October 2020 (6.11.2)

Contents

Chapter 1

Pivotal Greenplum 6.11 Release Notes

This document contains pertinent release information about Pivotal Greenplum 6.11 releases. For previous versions of the release notes for Greenplum Database, go to *Pivotal Greenplum Database Documentation*. For information about Greenplum Database end of life, see *Pivotal Greenplum Database end of life policy*.

Pivotal Greenplum 6 software is available for download from *VMware Tanzu Network*.

Pivotal Greenplum 6 is based on the open source *Greenplum Database project* code.

> **Important:** VMware does **not** provide support for open source versions of Greenplum Database. Only Pivotal Greenplum is supported by VMware.

Release 6.11.2

Release Date: 2020-10-2

Pivotal Greenplum 6.11.2 is a maintenance release that includes changes and resolves several issues.

Changed Features

Greenplum Database 6.11.2 includes these changes:

- Pivotal GPText version 3.4.5 is included, which includes bug fixes. See the *GPText 3.4.5 Release Notes* for more information.
- Pivotal Greenplum-Spark connector version 2.0.0 is included, which includes feature changes and bug fixes. See the *Greenplum-Spark Connector 2.0.0 Release Notes* for more information.

Resolved Issues

Pivotal Greenplum 6.11.2 resolves these issues:

30549 - Management and Monitoring

Greenplum excluded externally-routable loopback addresses from replication entries, which caused utilities such as `gpinitstandby` and `gpaddmirrors` to fail. This problem has been resolved.

30795 - GPORCA

Fixed a problem where GPORCA did not utilize an index scan for certain subqueries, which could lead to poor performance for affected queries.

30878 - GPORCA

If a `CREATE TABLE .. AS` statement was used to create a table with non-legacy (jump consistent) hash algorithm distribution from a source table that used the legacy (modulo) hash algorithm, GPORCA would distribute the data according to the value of `gp_use_legacy_hashops`; however, it would set the table's distribution policy hash algorithm to the value of the original table. This could cause queries to give incorrect results if the distribution policy did not match the data distribution. This problem has been resolved.

30903 - Metrics Collector

Workfile entries were sometimes freed prematurely, which could lead to the `postmaster` process being reset on segments and failures in query execution. This problem has been resolved.

30928 - GPORCA

If `gp_use_legacy_hashops` was enabled, GPORCA could crash when generating the query plan for certain queries that included an aggregate. This problem has been resolved.

174812955 - Query Execution

When executing a long query that contained multi-byte characters, Greenplum could incorrectly truncate the query string (removing multi-byte characters) and, if `log_min_duration_statement` was set to 0, could subsequent write an invalid symbol to segment logs. This behavior could cause errors in `gp_toolkit` and Command Center. This problem has been resolved.

Release 6.11.1

Release Date: 2020-09-17

Pivotal Greenplum 6.11.1 is a maintenance release that includes changes and resolves several issues.

Changed Features

Greenplum Database 6.11.1 includes this change:

- Greenplum Platform Extension Framework (PXF) version 5.15.1 is included, which includes changes and bug fixes. Refer to the *PXF Release Notes* for more information on release content and to access the PXF documentation.

Resolved Issues

Pivotal Greenplum 6.11.1 resolves these issues:

30751, 173714727 - Query Optimizer

Resolves an issue where a correlated subquery that contained at least one left or right outer join caused the Greenplum Database master to crash when the server configuration parameter `optimizer_join_order` was set to `exhaustive2`.

30880 - gpload

Fixed a problem where `gpload` operations would fail if a table column name included capital letters or special characters.

30901 - GPORCA

For queries that included an outer ref in a subquery, such as `select * from foo where foo.a = (select foo.b from bar)`, GPORCA always used the results of the subquery after unnesting the outer reference. This could cause a crash or incorrect results if the subquery returned no rows, or if the subquery contained a projection with multiple values below the outer reference. To address this problem, all such queries now fall back to using the Postgres planner instead of GPORCA. Note that this behavior occurs for cases where GPORCA would have returned correct results, as well as for cases that could cause crashes or return incorrect results.

30913, 170824967 - gpfdists

A command that accessed an external table using the `gpfdists` protocol failed if the external table did not use an IP address when specifying a host system in the `LOCATION` clause of the external table definition. This issue is resolved in Greenplum 6.11.1.

174609237 - gpstart

`gpstart` was updated so that it does not attempt to start a standby master segment when that segment is unreachable, preventing an associated stack trace during startup.

Upgrading from Greenplum 6.x to Greenplum 6.11

Note: Greenplum 6 does not support direct upgrades from Greenplum 4 or Greenplum 5 releases, or from earlier Greenplum 6 Beta releases.

See *Upgrading from an Earlier Greenplum 6 Release* to upgrade your existing Greenplum 6.x software to Greenplum 6.11.0.

Release 6.11.0

Release Date: 2020-09-11

Pivotal Greenplum 6.11.0 is a minor release that includes changed features and resolves several issues.

Features

Greenplum Database 6.11.0 includes these new and changed features:

- GPORCA partition elimination has been enhanced to support a subset of lossy assignment casts that are order-preserving (increasing) functions, including `timestamp::date` and `float::int`. For example, GPORCA supports partition elimination when a partition column is defined with the `timestamp` datatype and the query contains a predicate such as `WHERE ts::date == '2020-05-10'` that performs a cast on the partitioned column (`ts`) to compare column data (a `timestamp`) to a `date`.
- PXF version 5.15.0 is included, which includes new and changed features and bug fixes. Refer to the *PXF Release Notes* for more information on release content and supported platforms, and to access the PXF documentation.
- Greenplum Command Center 6.3.0 and 4.11.0 are included, which include new workload management and other features, as well as bug fixes. See the Command Center *Release Notes* for more information.
- The DataDirect ODBC Drivers for Pivotal Greenplum were updated to version 07.16.0389 (B0562, U0408). This version introduces support for the following datatypes:

Greenplum Datatype	ODBC Datatype
`citext`	`SQL_LONGVARCHAR`
`float`	`SQL_REAL`
`tinyint`	`SQL_SMALLINT`
`wchar`	`SQL_CHAR`
`wvarchar`	`SQL_VARCHAR`

Resolved Issues

Pivotal Greenplum 6.11.0 resolves these issues:

30899 - Resource Groups

In some cases when running queries are managed by resource groups, Greenplum Database generated a PANIC when managing runaway queries (queries that use an excessive amount of memory) because of locking issues. This issue is resolved.

30877 - VACUUM

In some cases, running `VACUUM` returns `ERROR: found xmin <xid> from before relfrozenxid <frozen_xid>`. The error was caused when a previously run `VACUUM FULL` was interrupted and aborted on a query executor (QE) and corrupted catalog frozen XID information. This issue is resolved.

30870 - Segment Mirroring

In some cases, performing an incremental recovery of a Greenplum Database segment instance failed with the message `requested WAL segment has already been removed` because the recovery checkpoint was not created properly. This issue is resolved.

30858 - analyzedb

`analyzedb` failed if `analyzedb` attempted to update statistics for a set of tables and one of the tables was dropped and then recreated while `analyzedb` was running. `analyzedb` has been enhanced better handle the specified situation.

30845 - Query Execution

Under heavy load when running multiple queries, some queries randomly failed with the error `Error on receive from seg<ID>`. The error was caused when Greenplum Database encountered a divide by 0 error while managing the backend processes that are used to run queries on the segment instances. This issue is resolved.

30761 - Postgres Planner

In some cases, Greenplum Database generated a PANIC when a `DROP VIEW` command was cancelled from the Greenplum Command Center. The PANIC was generated when Greenplum Database did not correctly handle the visibility of the relation.

30721 - gpcheckcat

Resolved a problem where `gpcheckcat` would fail with `Missing or extraneous entries check` errors if the `gp_sparse_vector` extension was installed.

30637 - Query Optimizer

For some queries against partitioned tables, GPORCA did not perform partition elimination when a predicate that includes the partition column also performs an explicit cast. For example, GPORCA would not perform partition elimination when a partition column is defined with the `timestamp` datatype and the query contains a predicate such as `WHERE ts::date == '2020-05-10'` that performs a cast on the partitioned column (`ts`) to compare column data (a `timestamp`) to a `date`. GPORCA partition elimination has been improved to support the specified type of query. See *Features*.

10491 **- Postgres Planner**

For some queries that contain nested subqueries that do not specify a relation and also contain a nested `GROUP BY` clauses, Greenplum Database generated a PANIC. The PANIC was generated when Greenplum Database did not correctly manage the subquery correctly. This is an example of the specified type or query.

```
SELECT * FROM (SELECT * FROM (SELECT c1, SUM(c2) c2 FROM mytbl
 GROUP BY c1 ) t2 ) t3
  GROUP BY c2, ROLLUP((c1))
  ORDER BY 1, 2;
```

This issue is resolved.

10561 **- Server**

Greenplum Database does not support altering the datatype of a column defined as a distribution key or with a constraint. When attempting to change the datatype, the error message did not clearly indicate the cause. The error message has been altered to provide more information.

174505130 - Resource Groups

In some cases for a query managed by resource group, the resource group cancelled the query with the message `Canceling query because of high VMEM usage` because the resource group calculated the incorrect memory used by the query. This issue is resolved.

174353156 - Interconnect

In some cases when Greenplum Database uses proxies for interconnect communication (the server configuration parameter `gp_interconnect_type` is set to `proxy`), a Greenplum background worker process became an orphaned process after the postmaster process was terminated. This issue is resolved.

174205590 - Interconnect

When Greenplum Database uses proxies for interconnect communication (the server configuration parameter `gp_interconnect_type` is set to `proxy`), a query might have hung if the query contains multiple concurrent subplans running on the segment instances. The query hung when the Greenplum interconnect did not properly handle the communication among the concurrent subplans. This issue is resolved.

`174483149 - Cluster Management - gpinitsystem`

`gpinitsystem` now exports the `MASTER_DATA_DIRECTORY` environment variable before calling `gpconfig`, to avoid throwing warning messages when configuring system parameters on Greenplum Database appliances (DCA).

Upgrading from Greenplum 6.x to Greenplum 6.11

Note: Greenplum 6 does not support direct upgrades from Greenplum 4 or Greenplum 5 releases, or from earlier Greenplum 6 Beta releases.

See *Upgrading from an Earlier Greenplum 6 Release* to upgrade your existing Greenplum 6.x software to Greenplum 6.11.0.

Deprecated Features

Deprecated features will be removed in a future major release of Greenplum Database. Pivotal Greenplum 6.x deprecates:

- The `gpsys1` utility.
- The `analzyedb` option `--skip_root_stats` (deprecated since 6.2).

  If the option is specified, a warning is issued stating that the option will be ignored.
- The server configuration parameter `gp_statistics_use_fkeys` (deprecated since 6.2).
- The server configuration parameter `gp_ignore_error_table` (deprecated since 6.0).

  To avoid a Greenplum Database syntax error, set the value of this parameter to `true` when you run applications that execute `CREATE EXTERNAL TABLE` or `COPY` commands that include the now removed Greenplum Database 4.3.x `INTO ERROR TABLE` clause.
- Specifying => as an operator name in the `CREATE OPERATOR` command (deprecated since 6.0).
- The Greenplum external table C API (deprecated since 6.0).

  Any developers using this API are encouraged to use the new Foreign Data Wrapper API in its place.
- Commas placed between a `SUBPARTITION TEMPLATE` clause and its corresponding `SUBPARTITION BY` clause, and between consecutive `SUBPARTITION BY` clauses in a `CREATE TABLE` command (deprecated since 6.0).

  Using this undocumented syntax will generate a deprecation warning message.
- The timestamp format YYYYMMDDHH24MISS (deprecated since 6.0).

  This format could not be parsed unambiguously in previous Greenplum Database releases, and is not supported in PostgreSQL 9.4.
- The `createlang` and `droplang` utilities (deprecated since 6.0).
- The `pg_resqueue_status` system view (deprecated since 6.0).

  Use the `gp_toolkit.gp_resqueue_status` view instead.
- The `GLOBAL` and `LOCAL` modifiers when creating a temporary table with the `CREATE TABLE` and `CREATE TABLE AS` commands (deprecated since 6.0).

  These keywords are present for SQL standard compatibility, but have no effect in Greenplum Database.
- Using `WITH OIDS` or `oids=TRUE` to assign an OID system column when creating or altering a table (deprecated since 6.0).
- Allowing superusers to specify the `SQL_ASCII` encoding regardless of the locale settings (deprecated since 6.0).

  This choice may result in misbehavior of character-string functions when data that is not encoding-compatible with the locale is stored in the database.
- The @@@ text search operator (deprecated since 6.0).

  This operator is currently a synonym for the @@ operator.
- The unparenthesized syntax for option lists in the `VACUUM` command (deprecated since 6.0).

  This syntax requires that the options to the command be specified in a specific order.
- The `plain` `pgbouncer` authentication type (`auth_type` = `plain`) (deprecated since 4.x).

Migrating Data to Greenplum 6

Note: Greenplum 6 does not support direct upgrades from Greenplum 4 or Greenplum 5 releases, or from earlier Greenplum 6 Beta releases.

See *Migrating Data from Greenplum 4.3 or 5* for guidelines and considerations for migrating existing Greenplum data to Greenplum 6, using standard backup and restore procedures.

Known Issues and Limitations

Pivotal Greenplum 6 has these limitations:

- Upgrading a Greenplum Database 4 or 5 release, or Greenplum 6 Beta release, to Greenplum 6 is not supported.
- MADlib, GPText, and PostGIS are not yet provided for installation on Ubuntu systems.
- Greenplum 6 is not supported for installation on DCA systems.
- Greenplum for Kubernetes is not yet provided with this release.

The following table lists key known issues in Pivotal Greenplum 6.x.

Table 1: Key Known Issues in Pivotal Greenplum 6.x

Issue	Category	Description
N/A	Backup/Restore	Restoring the Greenplum Database backup for a table fails in Greenplum 6 versions earlier than version 6.10 when a replicated table has an inheritance relationship to/from another table that was assigned via an `ALTER TABLE ... INHERIT` statement after table creation. **Workaround:** Use the following SQL commands to determine if Greenplum Database includes any replicated tables that inherit from a parent table, or if there are replicated tables that are inherited by a child table: `SELECT inhrelid::regclass FROM pg_inherits, gp_distribution_policy dp WHERE inhrelid=dp.localoid AND dp.policytype='r'; SELECT inhparent::regclass FROM pg_inherits, gp_distribution_policy dp WHERE inhparent=dp.localoid AND dp.policytype='r';` If these queries return any tables, you may choose to run `gprestore` with the `--on-error-continue` flag to not fail the entire restore when this issue is hit. Or, you can specify the list of tables returned by the queries to the `--exclude-table-file` option to skip those tables during restore. You must recreate and repopulate the affected tables after restore.
N/A	Spark Connector	This version of Greenplum is not compatible with Greenplum-Spark Connector versions earlier than version 1.7.0, due to a change in how Greenplum handles distributed transaction IDs.
N/A	PXF	Starting in 6.x, Greenplum does not bundle cURL and instead loads the system-provided library. PXF requires cURL version 7.29.0 or newer. The officially-supported cURL for the CentOS 6.x and Red Hat Enterprise Linux 6.x operating systems is version 7.19.*. Greenplum Database 6 does not support running PXF on CentOS 6.x or RHEL 6.x due to this limitation. **Workaround:** Upgrade the operating system of your Greenplum Database 6 hosts to CentOS 7+ or RHEL 7+, which provides a cURL version suitable to run PXF.

Issue	Category	Description
29703	Loading Data from External Tables	Due to limitations in the Greenplum Database external table framework, Greenplum Database cannot log the following types of errors that it encounters while loading data: • data type parsing errors • unexpected value type errors • data type conversion errors • errors returned by native and user-defined functions `LOG ERRORS` returns error information for data exceptions only. When it encounters a parsing error, Greenplum terminates the load job, but it cannot log and propagate the error back to the user via `gp_read_error_log()`. **Workaround**: Clean the input data before loading it into Greenplum Database.
30594	Resource Management	Resource queue-related statistics may be inaccurate in certain cases. VMware recommends that you use the resource group resource management scheme that is available in Greenplum 6.
30522	Logging	Greenplum Database may write a `FATAL` message to the standby master or mirror log stating that *the database system is in recovery mode* when the instance is synchronizing with the master and Greenplum attempts to contact it before the operation completes. Ignore these messages and use `gpstate -f` output to determine if the standby successfully synchronized with the Greenplum master; the command returns `Sync state: sync` if it is synchronized.
30537	Postgres Planner	The Postgres Planner generates a very large query plan that causes out of memory issues for the following type of CTE (common table expression) query: the `WITH` clause of the CTE contains a partitioned table with a large number partitions, and the `WITH` reference is used in a subquery that joins another partitioned table. **Workaround:** If possible, use the GPORCA query optimizer. With the server configuration parameter `optimizer=on`, Greenplum Database attempts to use GPORCA for query planning and optimization when possible and falls back to the Postgres Planner when GPORCA cannot be used. Also, the specified type of query might require a long time to complete.
170824967	gpfidsts	For Greenplum Database 6.x, a command that accesses an external table that uses the `gpfdists` protocol fails if the external table does not use an IP address when specifying a host system in the `LOCATION` clause of the external table definition. This issue is resolved in Greenplum 6.11.1.
n/a	Materialized Views	By default, certain `gp_toolkit` views do not display data for materialized views. If you want to include this information in `gp_toolkit` view output, you must redefine a `gp_toolkit` internal view as described in *Including Data for Materialized Views*.
168957894	PXF	The PXF Hive Connector does not support using the `Hive*` profiles to access Hive transactional tables. **Workaround:** Use the PXF JDBC Connector to access Hive.

Issue	Category	Description
168548176	gpbackup	When using `gpbackup` to back up a Greenplum Database 5.7.1 or earlier 5.x release with resource groups enabled, `gpbackup` returns a column not found error for `t6.value AS memoryauditor`.
164791118	PL/R	PL/R cannot be installed using the deprecated `createlang` utility, and displays the error: `createlang: language installation failed: ERROR: no schema has been selected to create in` **Workaround:** Use `CREATE EXTENSION` to install PL/R, as described in the documentation.
N/A	Greenplum Client/Load Tools on Windows	The Greenplum Database client and load tools on Windows have not been tested with Active Directory Kerberos authentication.

Differences Compared to Open Source Greenplum Database

Pivotal Greenplum 6.x includes all of the functionality in the open source *Greenplum Database project* and adds:

- Product packaging and installation script
- Support for QuickLZ compression. QuickLZ compression is not provided in the open source version of Greenplum Database due to licensing restrictions.
- Support for data connectors:
  - Greenplum-Spark Connector
  - Greenplum-Informatica Connector
  - Greenplum-Kafka Integration
  - Greenplum Streaming Server
- Data Direct ODBC/JDBC Drivers
- `gpcopy` utility for copying or migrating objects between Greenplum systems
- Support for managing Greenplum Database using Pivotal Greenplum Command Center
- Support for full text search and text analysis using Pivotal GPText
- Greenplum backup plugin for DD Boost
- Backup/restore storage plugin API

Chapter 2

Installing and Upgrading Greenplum

Information about installing, configuring, and upgrading Greenplum Database software and configuring Greenplum Database host machines.

Platform Requirements

This topic describes the Pivotal Greenplum Database 6 platform and operating system software requirements.

> **Important:** Pivotal Support does **not** provide support for open source versions of Greenplum Database. Only Pivotal Greenplum Database is supported by Pivotal Support.

Operating Systems

Pivotal Greenplum 6 runs on the following operating system platforms:

- Red Hat Enterprise Linux 64-bit 7.x (See the following *Note*.)
- Red Hat Enterprise Linux 64-bit 6.x
- CentOS 64-bit 7.x
- CentOS 64-bit 6.x
- Ubuntu 18.04 LTS
- Oracle Linux 64-bit 7, using the Red Hat Compatible Kernel (RHCK)

> **Important:** Significant Greenplum Database performance degradation has been observed when enabling resource group-based workload management on RedHat 6.x and CentOS 6.x systems. This issue is caused by a Linux cgroup kernel bug. This kernel bug has been fixed in CentOS 7.x and Red Hat 7.x systems.
>
> If you use RedHat 6 and the performance with resource groups is acceptable for your use case, upgrade your kernel to version 2.6.32-696 or higher to benefit from other fixes to the cgroups implementation.
>
> **Note:** For Greenplum Database that is installed on Red Hat Enterprise Linux 7.x or CentOS 7.x prior to 7.3, an operating system issue might cause Greenplum Database that is running large workloads to hang in the workload. The Greenplum Database issue is caused by Linux kernel bugs.
>
> RHEL 7.3 and CentOS 7.3 resolves the issue.

Greenplum Database server supports TLS version 1.2 on RHEL/CentOS systems, and TLS version 1.3 on Ubuntu systems.

Software Dependencies

Greenplum Database 6 requires the following software packages on RHEL/CentOS 6/7 systems which are installed automatically as dependencies when you install the Pivotal Greenplum Database RPM package):

- apr

- apr-util
- bash
- bzip2
- curl
- krb5
- libcurl
- libevent (or libevent2 on RHEL/CentOS 6)
- libxml2
- libyaml
- zlib
- openldap
- openssh
- openssl
- openssl-libs (RHEL7/Centos7)
- perl
- readline
- rsync
- R
- sed (used by `gpinitsystem`)
- tar
- zip

Greenplum Database 6 client software requires these operating system packages:

- apr
- apr-util
- libyaml
- libevent

On Ubuntu systems, Greenplum Database 6 requires the following software packages, which are installed automatically as dependencies when you install Greenplum Database with the Debian package installer:

- libapr1
- libaprutil1
- bash
- bzip2
- krb5-multidev
- libcurl3-gnutls
- libcurl4
- libevent-2.1-6
- libxml2
- libyaml-0-2
- zlib1g
- libldap-2.4-2
- openssh-client
- openssh-client
- openssl
- perl
- readline
- rsync
- sed
- tar
- zip

- net-tools
- less
- iproute2

Greenplum Database 6 uses Python 2.7.12, which is included with the product installation (and not installed as a package dependency).

> **Important:** SSL is supported only on the Greenplum Database master host system. It cannot be used on the segment host systems.

> **Important:** For all Greenplum Database host systems, SELinux must be disabled. You should also disable firewall software, although firewall software can be enabled if it is required for security purposes. See *Disabling SELinux and Firewall Software*.

Java

Greenplum 6 supports these Java versions for PL/Java and PXF:

- Open JDK 8 or Open JDK 11, available from *AdoptOpenJDK*
- Oracle JDK 8 or Oracle JDK 11

Hardware and Network

The following table lists minimum recommended specifications for hardware servers intended to support Greenplum Database on Linux systems in a production environment. All host servers in your Greenplum Database system must have the same hardware and software configuration. Greenplum also provides hardware build guides for its certified hardware platforms. It is recommended that you work with a Greenplum Systems Engineer to review your anticipated environment to ensure an appropriate hardware configuration for Greenplum Database.

Table 2: Minimum Hardware Requirements

Minimum CPU	Any x86_64 compatible CPU
Minimum Memory	16 GB RAM per server
Disk Space Requirements	• 150MB per host for Greenplum installation • Approximately 300MB per segment instance for meta data • Appropriate free space for data with disks at no more than 70% capacity
Network Requirements	10 Gigabit Ethernet within the array NIC bonding is recommended when multiple interfaces are present Pivotal Greenplum can use either IPV4 or IPV6 protocols.

Storage

The only file system supported for running Greenplum Database is the XFS file system. All other file systems are explicitly *not* supported by Pivotal.

Greenplum Database is supported on network or shared storage if the shared storage is presented as a block device to the servers running Greenplum Database and the XFS file system is mounted on the block device. Network file systems are *not* supported. When using network or shared storage, Greenplum

Database mirroring must be used in the same way as with local storage, and no modifications may be made to the mirroring scheme or the recovery scheme of the segments.

Other features of the shared storage such as de-duplication and/or replication are not directly supported by Pivotal Greenplum Database, but may be used with support of the storage vendor as long as they do not interfere with the expected operation of Greenplum Database at the discretion of Pivotal.

Greenplum Database can be deployed to virtualized systems only if the storage is presented as block devices and the XFS file system is mounted for the storage of the segment directories.

Greenplum Database is supported on Amazon Web Services (AWS) servers using either Amazon instance store (Amazon uses the volume names `ephemeral[0-20]`) or Amazon Elastic Block Store (Amazon EBS) storage. If using Amazon EBS storage the storage should be RAID of Amazon EBS volumes and mounted with the XFS file system for it to be a supported configuration.

Data Domain Boost

Pivotal Greenplum 6.0.0 supports Data Domain Boost for backup on Red Hat Enterprise Linux. This table lists the versions of Data Domain Boost SDK and DDOS supported by Pivotal Greenplum 6.x.

Table 3: Data Domain Boost Compatibility

Pivotal Greenplum	Data Domain Boost	DDOS
6.x	3.3	6.1 (all versions) 6.0 (all versions)

Note: In addition to the DDOS versions listed in the previous table, Pivotal Greenplum supports all minor patch releases (fourth digit releases) later than the certified version.

Tools and Extensions Compatibility

Client Tools

Greenplum Database 6 releases a Clients tool package on various platforms that can be used to access Greenplum Database from a client system. The Greenplum 6 Clients tool package is supported on the following platforms:

- Red Hat Enterprise Linux x86_64 6.x (RHEL 6)
- Red Hat Enterprise Linux x86_64 7.x (RHEL 7)
- Ubuntu 18.04 LTS
- Windows 10 (32-bit and 64-bit)
- Windows 8 (32-bit and 64-bit)
- Windows Server 2012 (32-bit and 64-bit)
- Windows Server 2012 R2 (32-bit and 64-bit)
- Windows Server 2008 R2 (32-bit and 64-bit)

The Greenplum 6 Clients package includes the client and loader programs provided in the Greenplum 5 packages plus the addition of database/role/language commands and the Greenplum-Kafka Integration and Greenplum Streaming Server command utilities. Refer to *Greenplum Client and Loader Tools Package* for installation and usage details of the Greenplum 6 Client tools.

Extensions

This table lists the versions of the Pivotal Greenplum Extensions that are compatible with this release of Greenplum Database 6.

Table 4: Pivotal Greenplum 6 Extensions Compatibility

Component	Package Version	Additional Information
PL/Java	2.0.2	Supports Java 8 and 11.
Python Data Science Module Package	2.0.2	
PL/R	3.0.3	(CentOS) R 3.3.3 (Ubuntu) You install R 3.5.1+.
R Data Science Library Package	2.0.2	
PL/Container	2.1.2	
PL/Container Image for R	2.1.2	R 3.6.3
PL/Container Images for Python	2.1.2	Python 2.7.12 Python 3.7
PL/Container Beta	3.0.0-beta	
PL/Container Beta Image for R	3.0.0-beta	R 3.4.4
GreenplumR	1.1.0	Supports R 3.6+.
MADlib Machine Learning	1.17, 1.16	Support matrix at *MADlib FAQ*.
PostGIS Spatial and Geographic Objects	2.5.4+pivotal.3, 2.5.4+pivotal.2, 2.5.4+pivotal.1, 2.1.5+pivotal.2-2	

For information about the Oracle Compatibility Functions, see *Oracle Compatibility Functions*.

These Greenplum Database extensions are installed with Pivotal Greenplum Database

- Fuzzy String Match Extension
- PL/Python Extension
- pgcrypto Extension

Data Connectors

- Greenplum Platform Extension Framework (PXF) v5.15.0 - PXF, integrated with Greenplum Database 6, provides access to Hadoop, object store, and SQL external data stores. Refer to *Accessing External Data with PXF* in the *Greenplum Database Administrator Guide* for PXF configuration and usage information.
- Greenplum-Kafka Integration - The Pivotal Greenplum-Kafka Integration provides high speed, parallel data transfer from a Kafka cluster to a Pivotal Greenplum Database cluster for batch and streaming ETL operations. It requires Kafka version 0.11 or newer for exactly-once delivery assurance. Refer to the *Pivotal Greenplum-Kafka Integration* Documentation for more information about this feature.
- Greenplum Streaming Server v1.4.1 - The Pivotal Greenplum Streaming Server is an ETL tool that provides high speed, parallel data transfer from Informatica, Kafka, and custom client data sources to a

Pivotal Greenplum Database cluster. Refer to the *Pivotal Greenplum Streaming Server* Documentation for more information about this feature.

- Greenplum Informatica Connector v1.0.5 - The Pivotal Greenplum Informatica Connector supports high speed data transfer from an Informatica PowerCenter cluster to a Pivotal Greenplum Database cluster for batch and streaming ETL operations.
- Greenplum Spark Connector v1.6.2 - The Pivotal Greenplum Spark Connector supports high speed, parallel data transfer between Greenplum Database and an Apache Spark cluster using Spark's Scala API.
- Progress DataDirect JDBC Drivers v5.1.4.000223 - The Progress DataDirect JDBC drivers are compliant with the Type 4 architecture, but provide advanced features that define them as Type 5 drivers.
- Progress DataDirect ODBC Drivers v7.1.6 (07.16.0389) - The Progress DataDirect ODBC drivers enable third party applications to connect via a common interface to the Pivotal Greenplum Database system.

  Note: Pivotal Greenplum 6 does not support the ODBC driver for Cognos Analytics V11.

  Connecting to IBM Cognos software with an ODBC driver is not supported. Greenplum Database supports connecting to IBM Cognos software with the DataDirect JDBC driver for Pivotal Greenplum. This driver is available as a download from *Pivotal Network*.

GPText

Pivotal Greenplum Database 6 is compatible with Pivotal Greenplum Text version 3.3.1 and later. See the *Greenplum Text documentation* for additional compatibility information.

Greenplum Command Center

Pivotal Greenplum Database 6.8 and later are compatible only with Pivotal Greenplum Command Center 6.2 and later. See the *Greenplum Command Center documentation* for additional compatibility information.

Hadoop Distributions

Greenplum Database provides access to HDFS with the Greenplum Platform Extension Framework (PXF).

PXF can use Cloudera, Hortonworks Data Platform, MapR, and generic Apache Hadoop distributions. PXF bundles all of the JAR files on which it depends, including the following Hadoop libraries:

Table 5: PXF Hadoop Supported Platforms

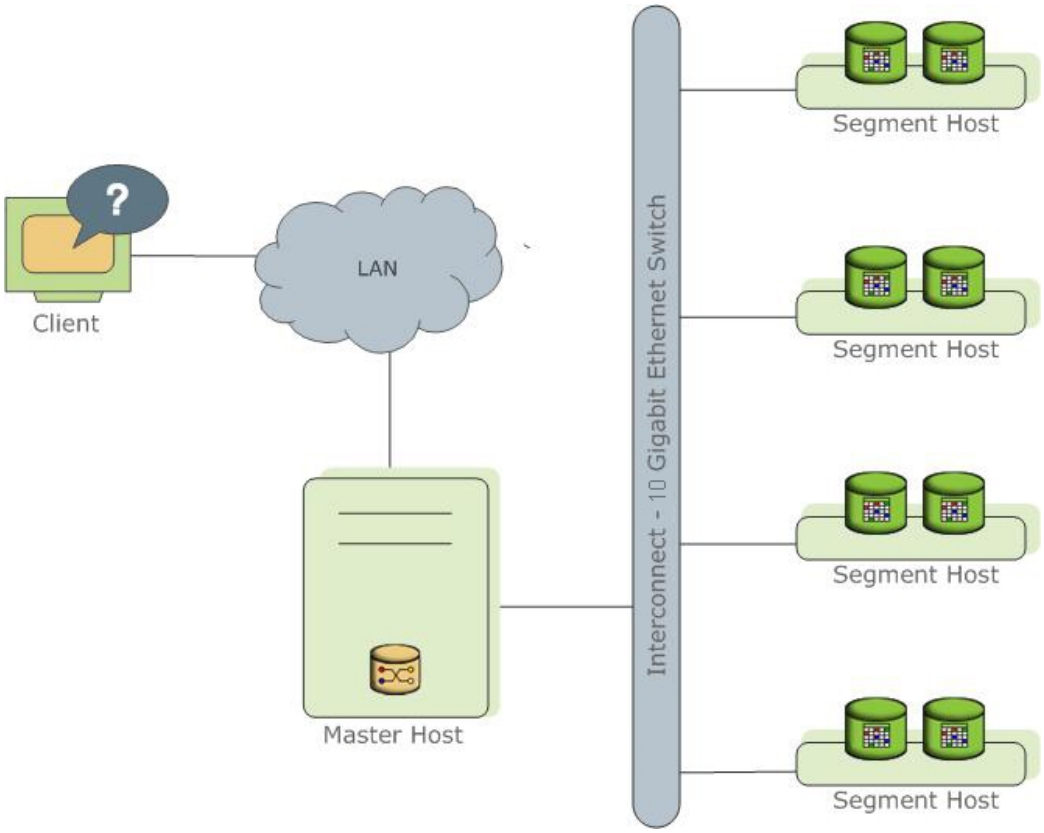
PXF Version	Hadoop Version	Hive Server Version	HBase Server Version
5.15.0, 5.14.0, 5.13.0, 5.12.0, 5.11.1, 5.10.1	2.x, 3.1+	1.x, 2.x, 3.1+	1.3.2
5.8.2	2.x	1.x	1.3.2
5.8.1	2.x	1.x	1.3.2

Note: If you plan to access JSON format data stored in a Cloudera Hadoop cluster, PXF requires a Cloudera version 5.8 or later Hadoop distribution.

Introduction to Greenplum

High-level overview of the Greenplum Database system architecture.

Greenplum Database stores and processes large amounts of data by distributing the load across several servers or *hosts*. A logical database in Greenplum is an *array* of individual PostgreSQL databases working together to present a single database image. The *master* is the entry point to the Greenplum Database system. It is the database instance to which users connect and submit SQL statements. The master coordinates the workload across the other database instances in the system, called *segments*, which handle data processing and storage. The segments communicate with each other and the master over the *interconnect*, the networking layer of Greenplum Database.



Greenplum Database is a software-only solution; the hardware and database software are not coupled. Greenplum Database runs on a variety of commodity server platforms from Greenplum-certified hardware vendors. Performance depends on the hardware on which it is installed. Because the database is distributed across multiple machines in a Greenplum Database system, proper selection and configuration of hardware is vital to achieving the best possible performance.

This chapter describes the major components of a Greenplum Database system and the hardware considerations and concepts associated with each component: *The Greenplum Master*, *The Segments* and *The Interconnect*. Additionally, a system may have optional *ETL Hosts for Data Loading* and *Greenplum Performance Monitoring* for monitoring query workload and performance.

The Greenplum Master

The *master* is the entry point to the Greenplum Database system. It is the database server process that accepts client connections and processes the SQL commands that system users issue. Users connect to Greenplum Database through the master using a PostgreSQL-compatible client program such as `psql` or ODBC.

The master maintains the *system catalog* (a set of system tables that contain metadata about the Greenplum Database system itself), however the master does not contain any user data. Data resides only on the *segments*. The master authenticates client connections, processes incoming SQL commands, distributes the work load between segments, coordinates the results returned by each segment, and presents the final results to the client program.

Because the master does not contain any user data, it has very little disk load. The master needs a fast, dedicated CPU for data loading, connection handling, and query planning because extra space is often necessary for landing load files and backup files, especially in production environments. Customers may decide to also run ETL and reporting tools on the master, which requires more disk space and processing power.

Master Redundancy

You may optionally deploy a *backup* or *mirror* of the master instance. A backup master host serves as a *warm standby* if the primary master host becomes nonoperational. You can deploy the standby master on a designated redundant master host or on one of the segment hosts.

The standby master is kept up to date by a transaction log replication process, which runs on the standby master host and synchronizes the data between the primary and standby master hosts. If the primary master fails, the log replication process shuts down, and an administrator can activate the standby master in its place. When an the standby master is active, the replicated logs are used to reconstruct the state of the master host at the time of the last successfully committed transaction.

Since the master does not contain any user data, only the system catalog tables need to be synchronized between the primary and backup copies. When these tables are updated, changes automatically copy over to the standby master so it is always synchronized with the primary.

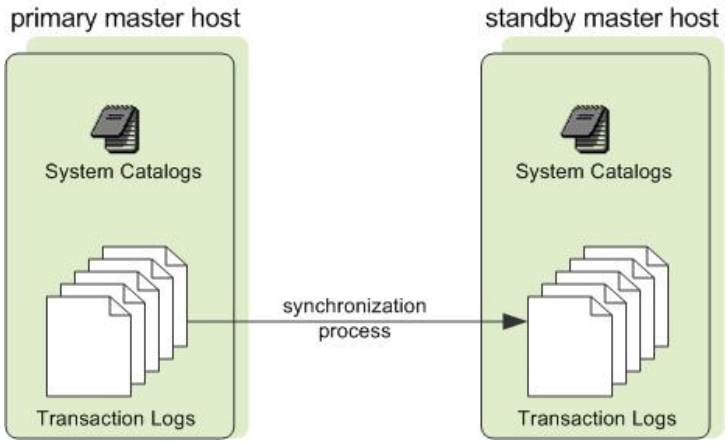


Figure 1: Master Mirroring in Greenplum Database

The Segments

In Greenplum Database, the *segments* are where data is stored and where most query processing occurs. User-defined tables and their indexes are distributed across the available segments in the Greenplum Database system; each segment contains a distinct portion of the data. Segment instances are the

database server processes that serve segments. Users do not interact directly with the segments in a Greenplum Database system, but do so through the master.

In the reference Greenplum Database hardware configurations, the number of segment instances per segment host is determined by the number of effective CPUs or CPU core. For example, if your segment hosts have two dual-core processors, you may have two or four primary segments per host. If your segment hosts have three quad-core processors, you may have three, six or twelve segments per host. Performance testing will help decide the best number of segments for a chosen hardware platform.

Segment Redundancy

When you deploy your Greenplum Database system, you have the option to configure *mirror* segments. Mirror segments allow database queries to fail over to a backup segment if the primary segment becomes unavailable. Mirroring is a requirement for Pivotal-supported production Greenplum Database systems.

A mirror segment must always reside on a different host than its primary segment. Mirror segments can be arranged across the hosts in the system in one of two standard configurations, or in a custom configuration you design. The default configuration, called *group* mirroring, places the mirror segments for all primary segments on a host on one other host. Another option, called *spread* mirroring, spreads mirrors for each host's primary segments over the remaining hosts. Spread mirroring requires that there be more hosts in the system than there are primary segments on the host. On hosts with multiple network interfaces, the primary and mirror segments are distributed equally among the interfaces. *Figure 2: Data Mirroring in Greenplum Database* shows how table data is distributed across the segments when the default group mirroring option is configured.

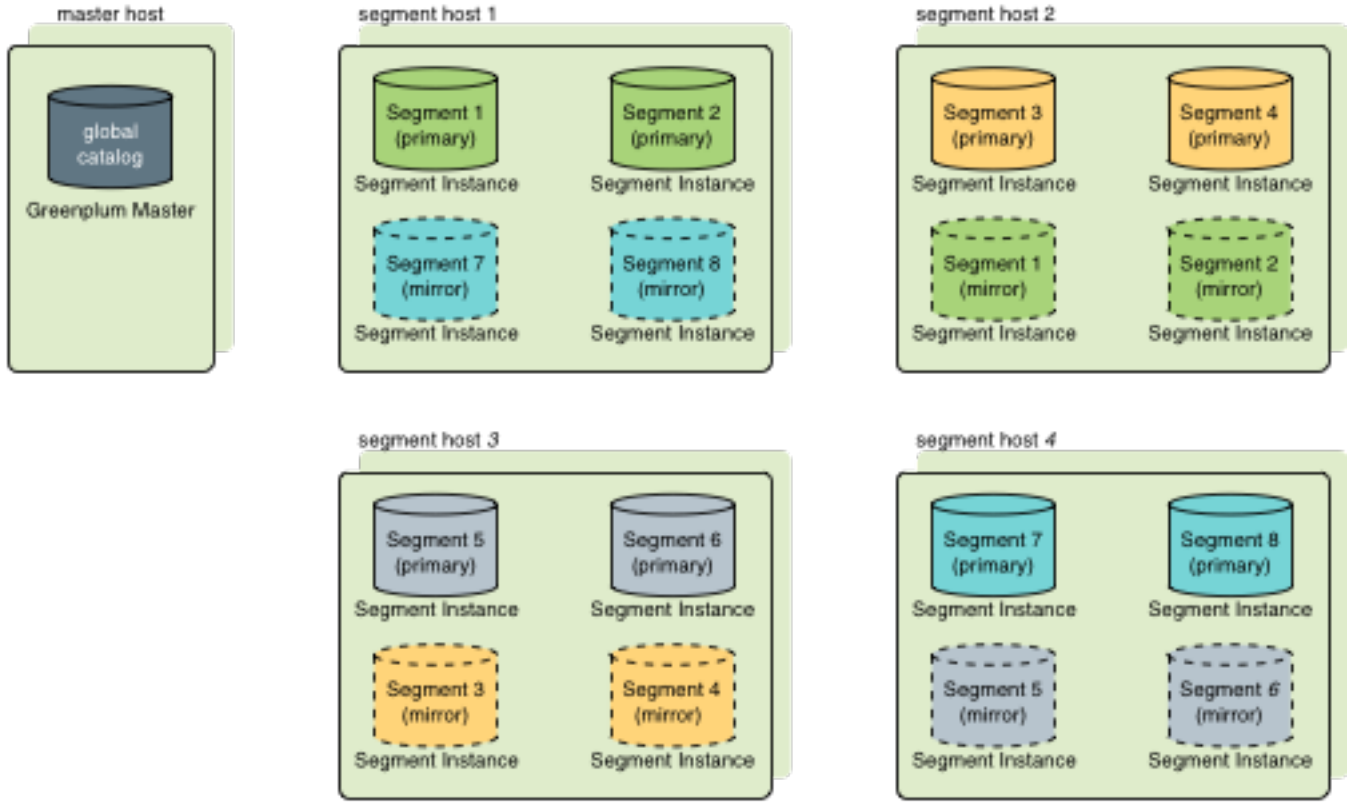


Figure 2: Data Mirroring in Greenplum Database

Segment Failover and Recovery

When mirroring is enabled in a Greenplum Database system, the system automatically fails over to the mirror copy if a primary copy becomes unavailable. A Greenplum Database system can remain operational

if a segment instance or host goes down only if all portions of data are available on the remaining active segments.

If the master cannot connect to a segment instance, it marks that segment instance as *invalid* in the Greenplum Database system catalog. The segment instance remains invalid and out of operation until an administrator brings that segment back online. An administrator can recover a failed segment while the system is up and running. The recovery process copies over only the changes that were missed while the segment was nonoperational.

If you do not have mirroring enabled and a segment becomes invalid, the system automatically shuts down. An administrator must recover all failed segments before operations can continue.

Example Segment Host Hardware Stack

Regardless of the hardware platform you choose, a production Greenplum Database processing node (a segment host) is typically configured as described in this section.

The segment hosts do the majority of database processing, so the segment host servers are configured in order to achieve the best performance possible from your Greenplum Database system. Greenplum Database's performance will be as fast as the slowest segment server in the array. Therefore, it is important to ensure that the underlying hardware and operating systems that are running Greenplum Database are all running at their optimal performance level. It is also advised that all segment hosts in a Greenplum Database array have identical hardware resources and configurations.

Segment hosts should also be dedicated to Greenplum Database operations only. To get the best query performance, you do not want Greenplum Database competing with other applications for machine or network resources.

The following diagram shows an example Greenplum Database segment host hardware stack. The number of effective CPUs on a host is the basis for determining how many primary Greenplum Database segment instances to deploy per segment host. This example shows a host with two effective CPUs (one dual-core CPU). Note that there is one primary segment instance (or primary/mirror pair if using mirroring) per CPU core.

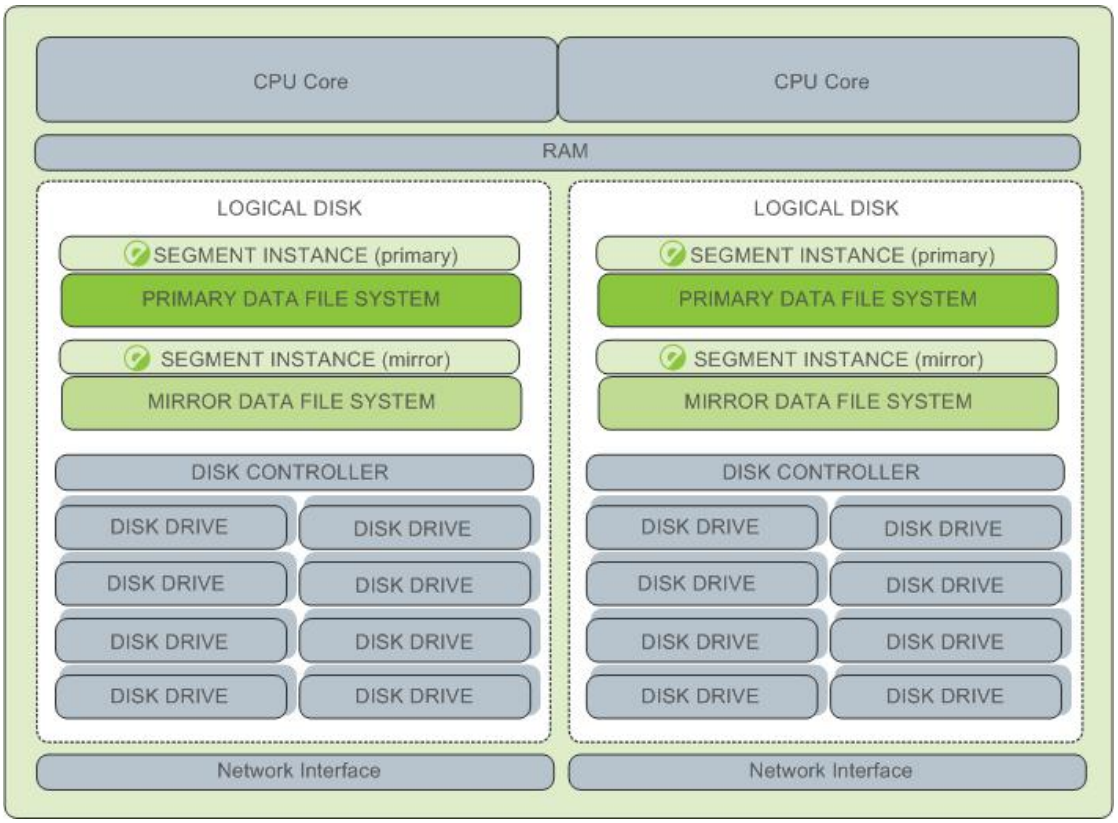


Figure 3: Example Greenplum Database Segment Host Configuration

Example Segment Disk Layout

Each CPU is typically mapped to a logical disk. A logical disk consists of one primary file system (and optionally a mirror file system) accessing a pool of physical disks through an I/O channel or disk controller. The logical disk and file system are provided by the operating system. Most operating systems provide the ability for a logical disk drive to use groups of physical disks arranged in RAID arrays.

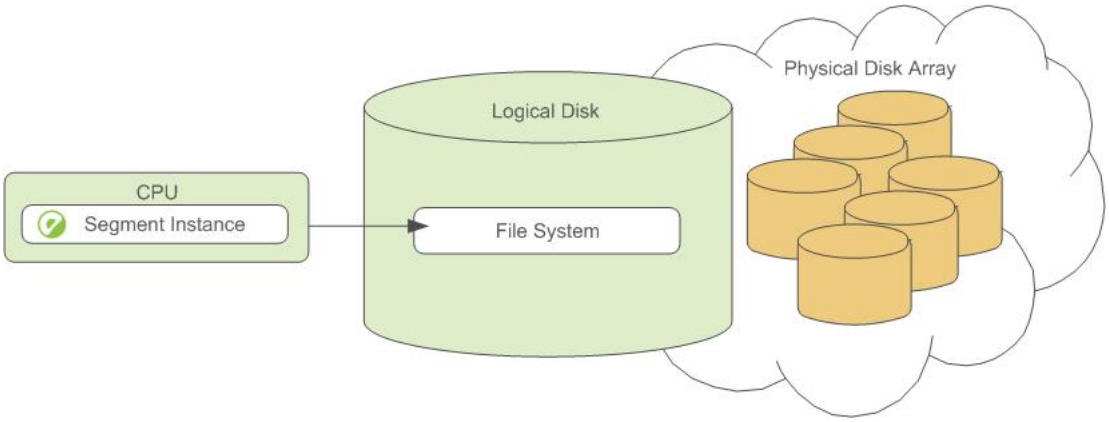


Figure 4: Logical Disk Layout in Greenplum Database

Depending on the hardware platform you choose, different RAID configurations offer different performance and capacity levels. Greenplum supports and certifies a number of reference hardware platforms and operating systems. Check with your sales account representative for the recommended configuration on your chosen platform.

The Interconnect

The *interconnect* is the networking layer of Greenplum Database. When a user connects to a database and issues a query, processes are created on each of the segments to handle the work of that query. The *interconnect* refers to the inter-process communication between the segments, as well as the network infrastructure on which this communication relies. The interconnect uses a standard 10 Gigabit Ethernet switching fabric.

By default, Greenplum Database interconnect uses UDP (User Datagram Protocol) with flow control for interconnect traffic to send messages over the network. The Greenplum software does the additional packet verification and checking not performed by UDP, so the reliability is equivalent to TCP (Transmission Control Protocol), and the performance and scalability exceeds that of TCP. For information about the types of interconnect supported by Greenplum Database, see server configuration parameter `gp_interconnect_type` in the *Greenplum Database Reference Guide*.

Interconnect Redundancy

A highly available interconnect can be achieved by deploying dual 10 Gigabit Ethernet switches on your network, and redundant 10 Gigabit connections to the Greenplum Database master and segment host servers.

Network Interface Configuration

A segment host typically has multiple network interfaces designated to Greenplum interconnect traffic. The master host typically has additional external network interfaces in addition to the interfaces used for interconnect traffic.

Depending on the number of interfaces available, you will want to distribute interconnect network traffic across the number of available interfaces. This is done by assigning segment instances to a particular network interface and ensuring that the primary segments are evenly balanced over the number of available interfaces.

This is done by creating separate host address names for each network interface. For example, if a host has four network interfaces, then it would have four corresponding host addresses, each of which maps to one or more primary segment instances. The `/etc/hosts` file should be configured to contain not only the host name of each machine, but also all interface host addresses for all of the Greenplum Database hosts (master, standby master, segments, and ETL hosts).

With this configuration, the operating system automatically selects the best path to the destination. Greenplum Database automatically balances the network destinations to maximize parallelism.

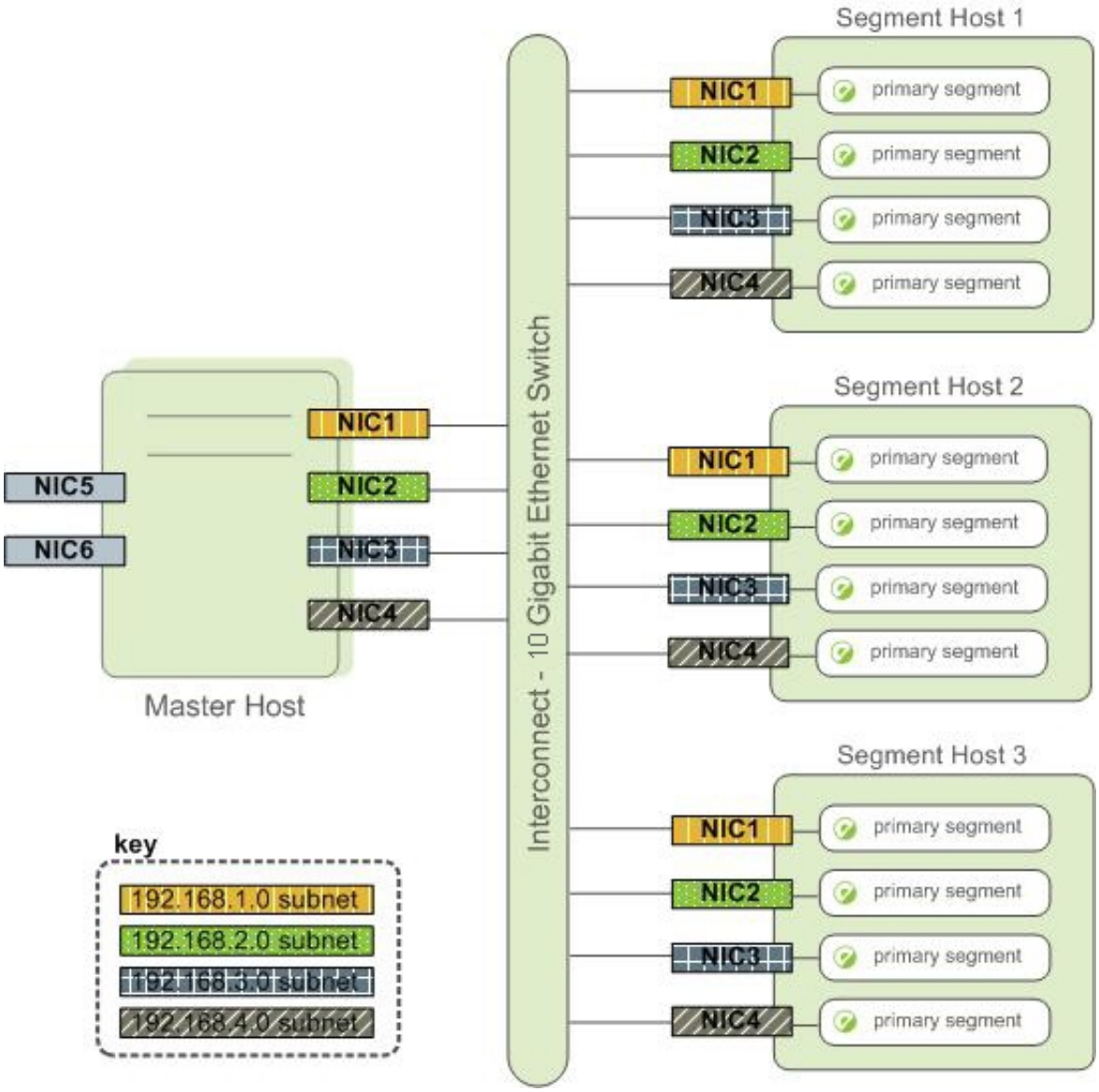


Figure 5: Example Network Interface Architecture

Switch Configuration

When using multiple 10 Gigabit Ethernet switches within your Greenplum Database array, evenly divide the number of subnets between each switch. In this example configuration, if we had two switches, NICs 1 and 2 on each host would use switch 1 and NICs 3 and 4 on each host would use switch 2. For the master host, the host name bound to NIC 1 (and therefore using switch 1) is the effective master host name for the array. Therefore, if deploying a warm standby master for redundancy purposes, the standby master should map to a NIC that uses a different switch than the primary master.

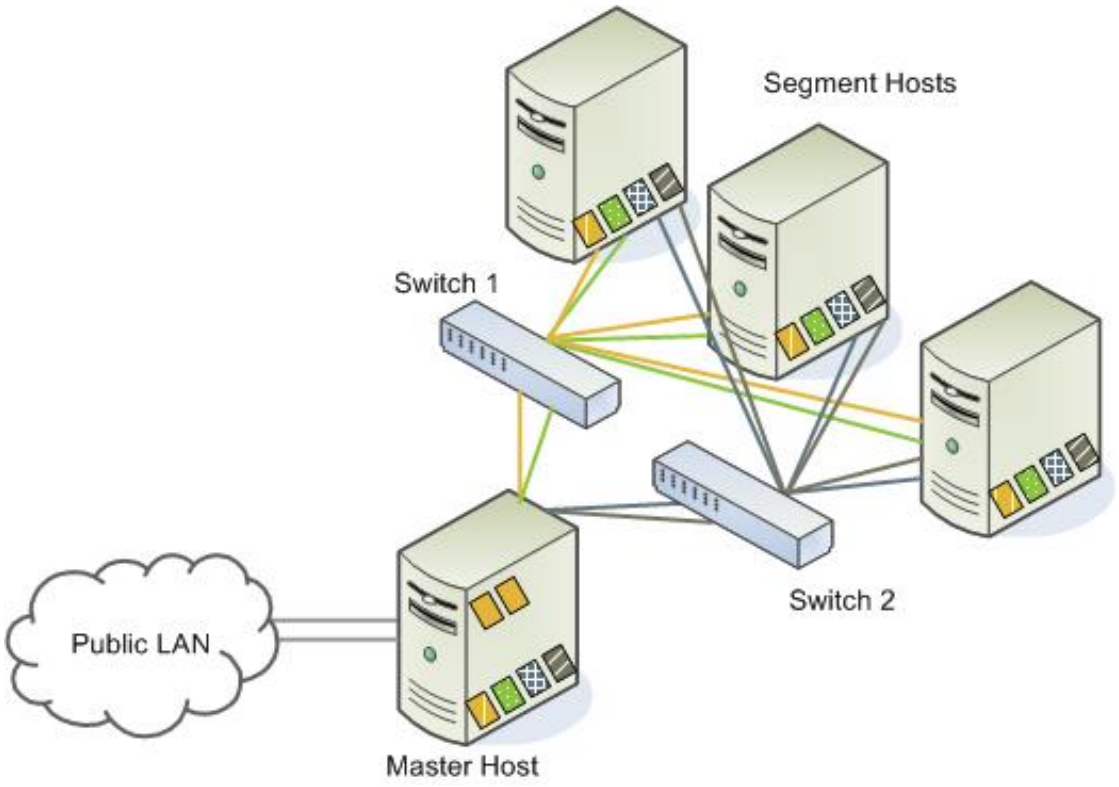


Figure 6: Example Switch Configuration

ETL Hosts for Data Loading

Greenplum supports fast, parallel data loading with its external tables feature. By using external tables in conjunction with Greenplum Database's parallel file server (`gpfdist`), administrators can achieve maximum parallelism and load bandwidth from their Greenplum Database system. Many production systems deploy designated ETL servers for data loading purposes. These machines run the Greenplum parallel file server (`gpfdist`), but not Greenplum Database instances.

One advantage of using the `gpfdist` file server program is that it ensures that all of the segments in your Greenplum Database system are fully utilized when reading from external table data files.

The `gpfdist` program can serve data to the segment instances at an average rate of about 350 MB/s for delimited text formatted files and 200 MB/s for CSV formatted files. Therefore, you should consider the following options when running `gpfdist` in order to maximize the network bandwidth of your ETL systems:

- If your ETL machine is configured with multiple network interface cards (NICs) as described in *Network Interface Configuration*, run one instance of `gpfdist` on your ETL host and then define your external table definition so that the host name of each NIC is declared in the `LOCATION` clause (see `CREATE EXTERNAL TABLE` in the *Greenplum Database Reference Guide*). This allows network traffic between your Greenplum segment hosts and your ETL host to use all NICs simultaneously.

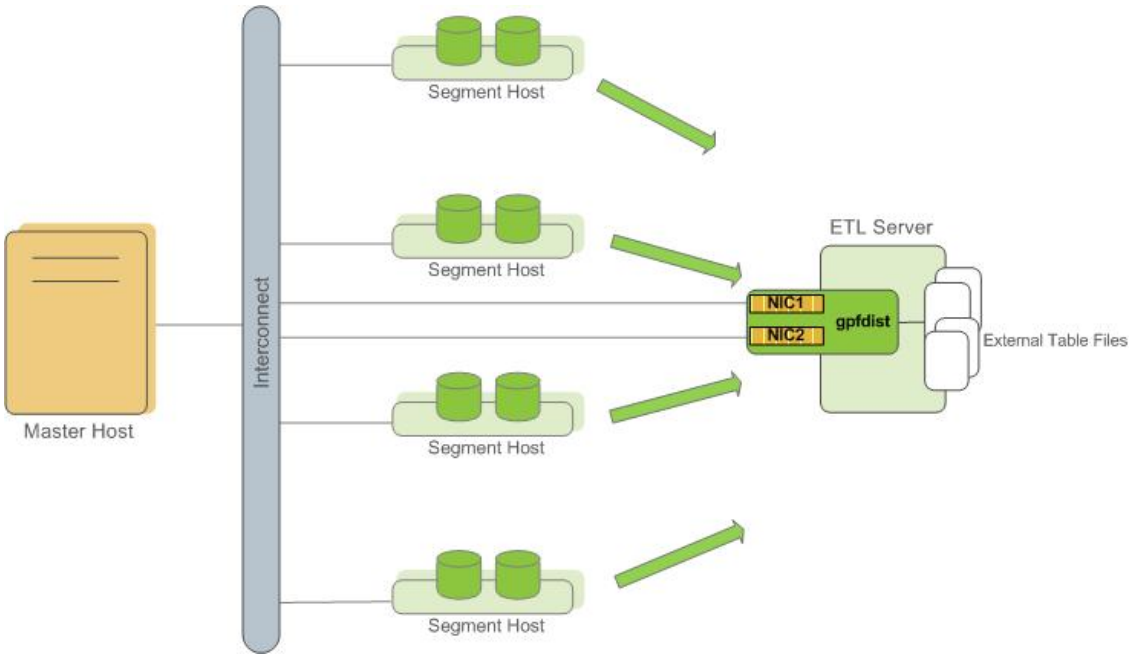


Figure 7: External Table Using Single gpfdist Instance with Multiple NICs

- Run multiple `gpfdist` instances on your ETL host and divide your external data files equally between each instance. For example, if you have an ETL system with two network interface cards (NICs), then you could run two `gpfdist` instances on that machine to maximize your load performance. You would then divide the external table data files evenly between the two `gpfdist` programs.

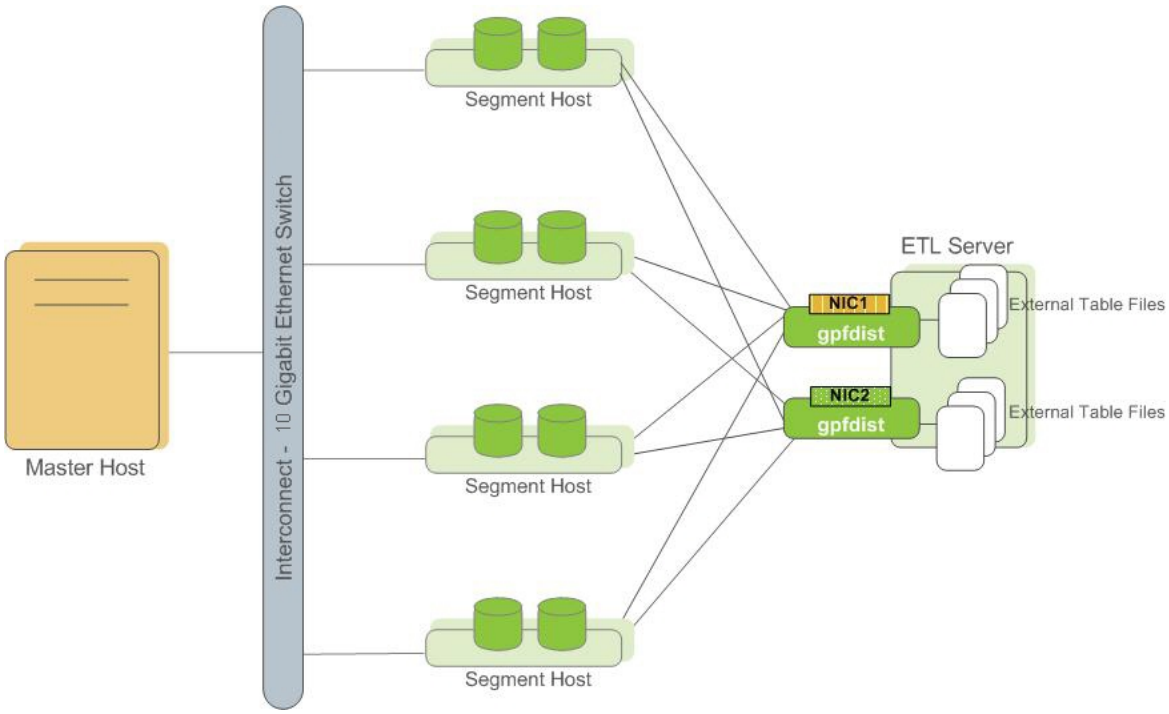


Figure 8: External Tables Using Multiple gpfdist Instances with Multiple NICs

Greenplum Performance Monitoring

Greenplum Database includes a dedicated system monitoring and management database, named gpperfmon, that administrators can install and enable. When this database is enabled, data collection

agents on each segment host collect query status and system metrics. At regular intervals (typically every 15 seconds), an agent on the Greenplum master requests the data from the segment agents and updates the gpperfmon database. Users can query the gpperfmon database to see the stored query and system metrics. For more information see the "gpperfmon Database Reference" in the *Greenplum Database Reference Guide*.

Greenplum Command Center is an optional web-based performance monitoring and management tool for Greenplum Database, based on the gpperfmon database. Administrators can install Command Center separately from Greenplum Database.

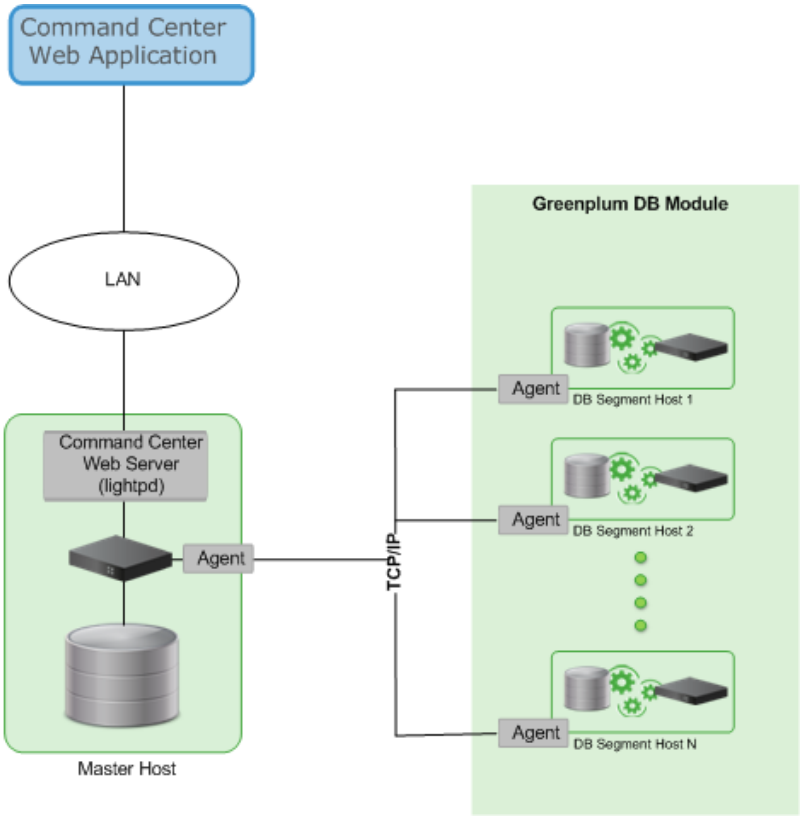


Figure 9: Greenplum Performance Monitoring Architecture

Estimating Storage Capacity

To estimate how much data your Greenplum Database system can accommodate, use these measurements as guidelines. Also keep in mind that you may want to have extra space for landing backup files and data load files on each segment host.

Calculating Usable Disk Capacity

To calculate how much data a Greenplum Database system can hold, you have to calculate the usable disk capacity per segment host and then multiply that by the number of segment hosts in your Greenplum Database array. Start with the raw capacity of the physical disks on a segment host that are available for data storage (*raw_capacity*), which is:

```
disk_size * number_of_disks
```

Account for file system formatting overhead (roughly 10 percent) and the RAID level you are using. For example, if using RAID-10, the calculation would be:

```
(raw_capacity * 0.9) / 2 = formatted_disk_space
```

For optimal performance, do not completely fill your disks to capacity, but run at 70% or lower. So with this in mind, calculate the usable disk space as follows:

```
formatted_disk_space * 0.7 = usable_disk_space
```

Using only 70% of your disk space allows Greenplum Database to use the other 30% for temporary and transaction files on the same disks. If your host systems have a separate disk system that can be used for temporary and transaction files, you can specify a tablespace that Greenplum Database uses for the files. Moving the location of the files might improve performance depending on the performance of the disk system.

Once you have formatted RAID disk arrays and accounted for the maximum recommended capacity (*usable_disk_space*), you will need to calculate how much storage is actually available for user data (`U`). If using Greenplum Database mirrors for data redundancy, this would then double the size of your user data (`2 * U`). Greenplum Database also requires some space be reserved as a working area for active queries. The work space should be approximately one third the size of your user data (work space = `U/3`):

```
With mirrors: (2 * U) + U/3 = usable_disk_space

Without mirrors: U + U/3 = usable_disk_space
```

Guidelines for temporary file space and user data space assume a typical analytic workload. Highly concurrent workloads or workloads with queries that require very large amounts of temporary space can benefit from reserving a larger working area. Typically, overall system throughput can be increased while decreasing work area usage through proper workload management. Additionally, temporary space and user space can be isolated from each other by specifying that they reside on different tablespaces.

In the *Greenplum Database Administrator Guide*, see these topics:

- *Managing Performance* for information about workload management
- *Creating and Managing Tablespaces* for information about moving the location of temporary and transaction files
- *Monitoring System State* for information about monitoring Greenplum Database disk space usage

Calculating User Data Size

As with all databases, the size of your raw data will be slightly larger once it is loaded into the database. On average, raw data will be about 1.4 times larger on disk after it is loaded into the database, but could be smaller or larger depending on the data types you are using, table storage type, in-database compression, and so on.

- Page Overhead - When your data is loaded into Greenplum Database, it is divided into pages of 32KB each. Each page has 20 bytes of page overhead.
- Row Overhead - In a regular 'heap' storage table, each row of data has 24 bytes of row overhead. An 'append-optimized' storage table has only 4 bytes of row overhead.
- Attribute Overhead - For the data values itself, the size associated with each attribute value is dependent upon the data type chosen. As a general rule, you want to use the smallest data type possible to store your data (assuming you know the possible values a column will have).
- Indexes - In Greenplum Database, indexes are distributed across the segment hosts as is table data. The default index type in Greenplum Database is B-tree. Because index size depends on the number of unique values in the index and the data to be inserted, precalculating the exact size of an index is impossible. However, you can roughly estimate the size of an index using these formulas.

```
B-tree: unique_values * (data_type_size + 24 bytes)

Bitmap: (unique_values * number_of_rows * 1 bit * compression_ratio / 8) +
 (unique_values * 32)
```

Calculating Space Requirements for Metadata and Logs

On each segment host, you will also want to account for space for Greenplum Database log files and metadata:

- **System Metadata** — For each Greenplum Database segment instance (primary or mirror) or master instance running on a host, estimate approximately 20 MB for the system catalogs and metadata.
- **Write Ahead Log** — For each Greenplum Database segment (primary or mirror) or master instance running on a host, allocate space for the write ahead log (WAL). The WAL is divided into segment files of 64 MB each. At most, the number of WAL files will be:

  ```
  2 * checkpoint_segments + 1
  ```

  You can use this to estimate space requirements for WAL. The default *checkpoint_segments* setting for a Greenplum Database instance is 8, meaning 1088 MB WAL space allocated for each segment or master instance on a host.
- **Greenplum Database Log Files** — Each segment instance and the master instance generates database log files, which will grow over time. Sufficient space should be allocated for these log files, and some type of log rotation facility should be used to ensure that to log files do not grow too large.
- **Command Center Data** — The data collection agents utilized by Command Center run on the same set of hosts as your Greenplum Database instance and utilize the system resources of those hosts. The resource consumption of the data collection agent processes on these hosts is minimal and should not significantly impact database performance. Historical data collected by the collection agents is stored in its own Command Center database (named `gpperfmon`) within your Greenplum Database system. Collected data is distributed just like regular database data, so you will need to account for disk space in the data directory locations of your Greenplum segment instances. The amount of space required depends on the amount of historical data you would like to keep. Historical data is not automatically truncated. Database administrators must set up a truncation policy to maintain the size of the Command Center database.

Configuring Your Systems

Describes how to prepare your operating system environment for Greenplum Database software installation.

Perform the following tasks in order:

1. Make sure your host systems meet the requirements described in *Platform Requirements*.
2. *Disable SELinux and firewall software.*
3. *Set the required operating system parameters.*
4. *Synchronize system clocks.*
5. *Create the gpadmin account.*

Unless noted, these tasks should be performed for *all* hosts in your Greenplum Database array (master, standby master, and segment hosts).

The Greenplum Database host naming convention for the master host is `mdw` and for the standby master host is `smdw`.

The segment host naming convention is *sdwN* where *sdw* is a prefix and *N* is an integer. For example, segment host names would be `sdw1`, `sdw2` and so on. NIC bonding is recommended for hosts with multiple interfaces, but when the interfaces are not bonded, the convention is to append a dash (`-`) and number to the host name. For example, `sdw1-1` and `sdw1-2` are the two interface names for host `sdw1`.

For information about running Greenplum Database in the cloud see *Cloud Services* in the *Pivotal Greenplum Partner Marketplace*.

> **Important:** When data loss is not acceptable for a Greenplum Database cluster, Greenplum master and segment mirroring is recommended. If mirroring is not enabled then Greenplum stores only one copy of the data, so the underlying storage media provides the only guarantee for data availability and correctness in the event of a hardware failure.
>
> Kubernetes enables quick recovery from both pod and host failures, and Kubernetes storage services provide a high level of availability for the underlying data. Furthermore, virtualized environments make it difficult to ensure the anti-affinity guarantees required for Greenplum mirroring solutions. For these reasons, mirrorless deployments are fully supported with Greenplum for Kubernetes. Other deployment environments are generally not supported for production use unless both Greenplum master and segment mirroring are enabled.
>
> **Note:** For information about upgrading Pivotal Greenplum Database from a previous version, see the *Greenplum Database Release Notes* for the release that you are installing.
>
> **Note:** Automating the configuration steps described in this topic and *Installing the Greenplum Database Software* with a system provisioning tool, such as Ansible, Chef, or Puppet, can save time and ensure a reliable and repeatable Greenplum Database installation.

Disabling SELinux and Firewall Software

For all Greenplum Database host systems running RHEL or CentOS, SELinux must be disabled. Follow these steps:

1. As the root user, check the status of SELinux:

```
# sestatus
SELinuxstatus: disabled
```

2. If SELinux is not disabled, disable it by editing the `/etc/selinux/config` file. As root, change the value of the `SELINUX` parameter in the `config` file as follows:

   ```
   SELINUX=disabled
   ```

3. If the System Security Services Daemon (SSSD) is installed on your systems, edit the SSSD configuration file and set the `selinux_provider` parameter to `none` to prevent SELinux-related SSH authentication denials that could occur even with SELinux disabled. As root, edit `/etc/sssd/sssd.conf` and add this parameter:

   ```
   selinux_provider=none
   ```

4. Reboot the system to apply any changes that you made and verify that SELinux is disabled.

For information about disabling SELinux, see the SELinux documentation.

You should also disable firewall software such as `iptables` (on systems such as RHEL 6.x and CentOS 6.x), `firewalld` (on systems such as RHEL 7.x and CentOS 7.x), or `ufw` (on Ubuntu systems, disabled by default).

If you decide to enable `iptables` with Greenplum Database for security purposes, see *Enabling iptables (Optional)*.

Follow these steps to disable `iptables`:

1. As the root user, check the status of `iptables`:

   ```
   # /sbin/chkconfig --list iptables
   ```

   If `iptables` is disabled, the command output is:

   ```
   iptables 0:off 1:off 2:off 3:off 4:off 5:off 6:off
   ```

2. If necessary, execute this command as root to disable `iptables`:

   ```
   /sbin/chkconfig iptables off
   ```

   You will need to reboot your system after applying the change.

3. For systems with `firewalld`, check the status of `firewalld` with the command:

   ```
   # systemctl status firewalld
   ```

   If `firewalld` is disabled, the command output is:

   ```
   * firewalld.service - firewalld - dynamic firewall daemon
      Loaded: loaded (/usr/lib/systemd/system/firewalld.service; disabled;
    vendor preset: enabled)
      Active: inactive (dead)
   ```

4. If necessary, execute these commands as root to disable `firewalld`:

   ```
   # systemctl stop firewalld.service
   # systemctl disable firewalld.service
   ```

For more information about configuring your firewall software, see the documentation for the firewall or your operating system.

Recommended OS Parameters Settings

Greenplum requires that certain Linux operating system (OS) parameters be set on all hosts in your Greenplum Database system (masters and segments).

In general, the following categories of system parameters need to be altered:

- **Shared Memory** - A Greenplum Database instance will not work unless the shared memory segment for your kernel is properly sized. Most default OS installations have the shared memory values set too low for Greenplum Database. On Linux systems, you must also disable the OOM (out of memory) killer. For information about Greenplum Database shared memory requirements, see the Greenplum Database server configuration parameter `shared_buffers` in the *Greenplum Database Reference Guide*.
- **Network** - On high-volume Greenplum Database systems, certain network-related tuning parameters must be set to optimize network connections made by the Greenplum interconnect.
- **User Limits** - User limits control the resources available to processes started by a user's shell. Greenplum Database requires a higher limit on the allowed number of file descriptors that a single process can have open. The default settings may cause some Greenplum Database queries to fail because they will run out of file descriptors needed to process the query.

More specifically, you need to edit the following Linux configuration settings:

- *The hosts File*
- *The sysctl.conf File*
- *System Resources Limits*
- *XFS Mount Options*
- *Disk I/O Settings*
  - Read ahead values
  - Disk I/O scheduler disk access
- *Transparent Huge Pages (THP)*
- *IPC Object Removal*
- *SSH Connection Threshold*

The `hosts` File

Edit the `/etc/hosts` file and make sure that it includes the host names and all interface address names for every machine participating in your Greenplum Database system.

The `sysctl.conf` File

The `sysctl.conf` parameters listed in this topic are for performance, optimization, and consistency in a wide variety of environments. Change these settings according to your specific situation and setup.

Set the parameters in the `/etc/sysctl.conf` file and reload with `sysctl -p`:

```
# kernel.shmall = _PHYS_PAGES / 2 # See Shared Memory Pages
kernel.shmall = 197951838
# kernel.shmmax = kernel.shmall * PAGE_SIZE
kernel.shmmax = 810810728448
kernel.shmmni = 4096
vm.overcommit_memory = 2 # See Segment Host Memory
vm.overcommit_ratio = 95 # See Segment Host Memory

net.ipv4.ip_local_port_range = 10000 65535 # See Port Settings
kernel.sem = 500 2048000 200 4096
kernel.sysrq = 1
kernel.core_uses_pid = 1
kernel.msgmnb = 65536
kernel.msgmax = 65536
kernel.msgmni = 2048
net.ipv4.tcp_syncookies = 1
net.ipv4.conf.default.accept_source_route = 0
net.ipv4.tcp_max_syn_backlog = 4096
```

```
net.ipv4.conf.all.arp_filter = 1
net.core.netdev_max_backlog = 10000
net.core.rmem_max = 2097152
net.core.wmem_max = 2097152
vm.swappiness = 10
vm.zone_reclaim_mode = 0
vm.dirty_expire_centisecs = 500
vm.dirty_writeback_centisecs = 100
vm.dirty_background_ratio = 0 # See System Memory
vm.dirty_ratio = 0
vm.dirty_background_bytes = 1610612736
vm.dirty_bytes = 4294967296
```

Shared Memory Pages

Greenplum Database uses shared memory to communicate between `postgres` processes that are part of the same `postgres` instance. `kernel.shmall` sets the total amount of shared memory, in pages, that can be used system wide. `kernel.shmmax` sets the maximum size of a single shared memory segment in bytes.

Set `kernel.shmall` and `kernel.shmax` values based on your system's physical memory and page size. In general, the value for both parameters should be one half of the system physical memory.

Use the operating system variables _PHYS_PAGES and PAGE_SIZE to set the parameters.

```
kernel.shmall = ( _PHYS_PAGES / 2)
kernel.shmmax = ( _PHYS_PAGES / 2) * PAGE_SIZE
```

To calculate the values for `kernel.shmall` and `kernel.shmax`, run the following commands using the `getconf` command, which returns the value of an operating system variable.

```
$ echo $(expr $(getconf _PHYS_PAGES) / 2)
$ echo $(expr $(getconf _PHYS_PAGES) / 2 \* $(getconf PAGE_SIZE))
```

As best practice, we recommend you set the following values in the `/etc/sysctl.conf` file using calculated values. For example, a host system has 1583 GB of memory installed and returns these values: _PHYS_PAGES = 395903676 and PAGE_SIZE = 4096. These would be the `kernel.shmall` and `kernel.shmmax` values:

```
kernel.shmall = 197951838
kernel.shmmax = 810810728448
```

If the Greeplum Database master the has a different shared memory configuration than the segment hosts, the _PHYS_PAGES and PAGE_SIZE values might differ, and the `kernel.shmall` and `kernel.shmax` values on the master host will differ from those on the segment hosts.

Segment Host Memory

The `vm.overcommit_memory` Linux kernel parameter is used by the OS to determine how much memory can be allocated to processes. For Greenplum Database, this parameter should always be set to 2.

`vm.overcommit_ratio` is the percent of RAM that is used for application processes and the remainder is reserved for the operating system. The default is 50 on Red Hat Enterprise Linux.

For `vm.overcommit_ratio` tuning and calculation recommendations with resource group-based resource management or resource queue-based resource management, refer to *Options for Configuring Segment Host Memory* in the *Geenplum Database Administrator Guide*. Also refer to the Greenplum Database server configuration parameter *gp_vmem_protect_limit* in the *Greenplum Database Reference Guide*.

Port Settings

To avoid port conflicts between Greenplum Database and other applications during Greenplum initialization, make a note of the port range specified by the operating system parameter `net.ipv4.ip_local_port_range`. When initializing Greenplum using the `gpinitsystem` cluster configuration file, do not specify Greenplum Database ports in that range. For example, if `net.ipv4.ip_local_port_range = 10000 65535`, set the Greenplum Database base port numbers to these values.

```
PORT_BASE = 6000
MIRROR_PORT_BASE = 7000
```

For information about the `gpinitsystem` cluster configuration file, see *Initializing a Greenplum Database System*.

For Azure deployments with Greenplum Database avoid using port 65330; add the following line to `sysctl.conf`:

```
net.ipv4.ip_local_reserved_ports=65330
```

For additional requirements and recommendations for cloud deployments, see *Greenplum Database Cloud Technical Recommendations*.

System Memory

For host systems with more than 64GB of memory, these settings are recommended:

```
vm.dirty_background_ratio = 0
vm.dirty_ratio = 0
vm.dirty_background_bytes = 1610612736 # 1.5GB
vm.dirty_bytes = 4294967296 # 4GB
```

For host systems with 64GB of memory or less, remove `vm.dirty_background_bytes` and `vm.dirty_bytes` and set the two `ratio` parameters to these values:

```
vm.dirty_background_ratio = 3
vm.dirty_ratio = 10
```

Increase `vm.min_free_kbytes` to ensure `PF_MEMALLOC` requests from network and storage drivers are easily satisfied. This is especially critical on systems with large amounts of system memory. The default value is often far too low on these systems. Use this awk command to set `vm.min_free_kbytes` to a recommended 3% of system physical memory:

```
awk 'BEGIN {OFMT = "%.0f";} /MemTotal/ {print "vm.min_free_kbytes =", $2
 * .03;}'
               /proc/meminfo >> /etc/sysctl.conf
```

Do not set `vm.min_free_kbytes` to higher than 5% of system memory as doing so might cause out of memory conditions.

System Resources Limits

Set the following parameters in the `/etc/security/limits.conf` file:

```
* soft nofile 524288
* hard nofile 524288
* soft nproc 131072
* hard nproc 131072
```

For Red Hat Enterprise Linux (RHEL) and CentOS systems, parameter values in the `/etc/security/limits.d/90-nproc.conf` file (RHEL/CentOS 6) or `/etc/security/limits.d/20-nproc.conf` file (RHEL/CentOS 7) override the values in the `limits.conf` file. Ensure that any parameters in the

override file are set to the required value. The Linux module `pam_limits` sets user limits by reading the values from the `limits.conf` file and then from the override file. For information about PAM and user limits, see the documentation on PAM and `pam_limits`.

Execute the `ulimit -u` command on each segment host to display the maximum number of processes that are available to each user. Validate that the return value is 131072.

XFS Mount Options

XFS is the preferred data storage file system on Linux platforms. Use the `mount` command with the following recommended XFS mount options for RHEL and CentOS systems:

```
rw,nodev,noatime,nobarrier,inode64
```

The `nobarrier` option is not supported on Ubuntu systems. Use only the options:

```
rw,nodev,noatime,inode64
```

See the `mount` manual page (`man mount` opens the man page) for more information about using this command.

The XFS options can also be set in the `/etc/fstab` file. This example entry from an `fstab` file specifies the XFS options.

```
/dev/data /data xfs nodev,noatime,nobarrier,inode64 0 0
```

Disk I/O Settings

- Read-ahead value

  Each disk device file should have a read-ahead (`blockdev`) value of 16384. To verify the read-ahead value of a disk device:

  ```
  # /sbin/blockdev --getra devname
  ```

  For example:

  ```
  # /sbin/blockdev --getra /dev/sdb
  ```

  To set blockdev (read-ahead) on a device:

  ```
  # /sbin/blockdev --setra bytes devname
  ```

  For example:

  ```
  # /sbin/blockdev --setra 16384 /dev/sdb
  ```

  See the manual page (man) for the `blockdev` command for more information about using that command (`man blockdev` opens the man page).

  > **Note:** The `blockdev --setra` command is not persistent. You must ensure the read-ahead value is set whenever the system restarts. How to set the value will vary based on your system.

  One method to set the `blockdev` value at system startup is by adding the `/sbin/blockdev --setra` command in the `rc.local` file. For example, add this line to the `rc.local` file to set the read-ahead value for the disk `sdb`.

  ```
  /sbin/blockdev --setra 16384 /dev/sdb
  ```

On systems that use systemd, you must also set the execute permissions on the `rc.local` file to enable it to run at startup. For example, on a RHEL/CentOS 7 system, this command sets execute permissions on the file.

```
# chmod +x /etc/rc.d/rc.local
```

Restart the system to have the setting take effect.

- Disk I/O scheduler

  The Linux disk I/O scheduler for disk access supports different policies, such as `CFQ`, `AS`, and `deadline`.

  The `deadline` scheduler option is recommended. To specify a scheduler until the next system reboot, run the following:

  ```
  # echo schedulername > /sys/block/devname/queue/scheduler
  ```

  For example:

  ```
  # echo deadline > /sys/block/sbd/queue/scheduler
  ```

  > **Note:** Using the `echo` command to set the disk I/O scheduler policy is not persistent, therefore you must ensure the command is run whenever the system reboots. How to run the command will vary based on your system.

  One method to set the I/O scheduler policy at boot time is with the `elevator` kernel parameter. Add the parameter `elevator=deadline` to the `kernel` command in the file `/boot/grub/grub.conf`, the GRUB boot loader configuration file. This is an example `kernel` command from a `grub.conf` file on RHEL 6.x or CentOS 6.x. The command is on multiple lines for readability.

  ```
  kernel /vmlinuz-2.6.18-274.3.1.el5 ro root=LABEL=/
                    elevator=deadline crashkernel=128M@16M  quiet
   console=tty1
                    console=ttyS1,115200 panic=30 transparent_hugepage=never
                    initrd /initrd-2.6.18-274.3.1.el5.img
  ```

  To specify the I/O scheduler at boot time on systems that use `grub2` such as RHEL 7.x or CentOS 7.x, use the system utility `grubby`. This command adds the parameter when run as root.

  ```
  # grubby --update-kernel=ALL --args="elevator=deadline"
  ```

  After adding the parameter, reboot the system.

  This `grubby` command displays kernel parameter settings.

  ```
  # grubby --info=ALL
  ```

  For more information about the `grubby` utility, see your operating system documentation. If the `grubby` command does not update the kernels, see the *Note* at the end of the section.

Transparent Huge Pages (THP)

Disable Transparent Huge Pages (THP) as it degrades Greenplum Database performance. RHEL 6.0 or higher enables THP by default. One way to disable THP on RHEL 6.x is by adding the parameter `transparent_hugepage=never` to the `kernel` command in the file `/boot/grub/grub.conf`, the GRUB boot loader configuration file. This is an example `kernel` command from a `grub.conf` file. The command is on multiple lines for readability:

```
kernel /vmlinuz-2.6.18-274.3.1.el5 ro root=LABEL=/
```

```
            elevator=deadline crashkernel=128M@16M  quiet console=tty1
            console=ttyS1,115200 panic=30 transparent_hugepage=never
            initrd /initrd-2.6.18-274.3.1.el5.img
```

On systems that use `grub2` such as RHEL 7.x or CentOS 7.x, use the system utility `grubby`. This command adds the parameter when run as root.

```
# grubby --update-kernel=ALL --args="transparent_hugepage=never"
```

After adding the parameter, reboot the system.

For Ubuntu systems, install the `hugepages` package and execute this command as root:

```
# hugeadm --thp-never
```

This `cat` command checks the state of THP. The output indicates that THP is disabled.

```
$ cat /sys/kernel/mm/*transparent_hugepage/enabled
always [never]
```

For more information about Transparent Huge Pages or the `grubby` utility, see your operating system documentation. If the `grubby` command does not update the kernels, see the *Note* at the end of the section.

IPC Object Removal

Disable IPC object removal for RHEL 7.2 or CentOS 7.2, or Ubuntu. The default `systemd` setting `RemoveIPC=yes` removes IPC connections when non-system user accounts log out. This causes the Greenplum Database utility `gpinitsystem` to fail with semaphore errors. Perform one of the following to avoid this issue.

- When you add the `gpadmin` operating system user account to the master node in *Creating the Greenplum Administrative User*, create the user as a system account.
- Disable `RemoveIPC`. Set this parameter in `/etc/systemd/logind.conf` on the Greenplum Database host systems.

  ```
  RemoveIPC=no
  ```

  The setting takes effect after restarting the `systemd-login` service or rebooting the system. To restart the service, run this command as the root user.

  ```
  service systemd-logind restart
  ```

SSH Connection Threshold

Certain Greenplum Database management utilities including `gpexpand`, `gpinitsystem`, and `gpaddmirrors`, use secure shell (SSH) connections between systems to perform their tasks. In large Greenplum Database deployments, cloud deployments, or deployments with a large number of segments per host, these utilities may exceed the hosts' maximum threshold for unauthenticated connections. When this occurs, you receive errors such as: `ssh_exchange_identification: Connection closed by remote host`.

To increase this connection threshold for your Greenplum Database system, update the SSH `MaxStartups` and `MaxSessions` configuration parameters in one of the `/etc/ssh/sshd_config` or `/etc/sshd_config` SSH daemon configuration files.

If you specify `MaxStartups` and `MaxSessions` using a single integer value, you identify the maximum number of concurrent unauthenticated connections (`MaxStartups`) and maximum number of open shell, login, or subsystem sessions permitted per network connection (`MaxSessions`). For example:

```
MaxStartups 200
MaxSessions 200
```

If you specify `MaxStartups` using the "*start:rate:full*" syntax, you enable random early connection drop by the SSH daemon. *start* identifies the maximum number of unauthenticated SSH connection attempts allowed. Once *start* number of unauthenticated connection attempts is reached, the SSH daemon refuses *rate* percent of subsequent connection attempts. *full* identifies the maximum number of unauthenticated connection attempts after which all attempts are refused. For example:

```
Max Startups 10:30:200
MaxSessions 200
```

Restart the SSH daemon after you update `MaxStartups` and `MaxSessions`. For example, on a CentOS 6 system, run the following command as the `root` user:

```
# service sshd restart
```

For detailed information about SSH configuration options, refer to the SSH documentation for your Linux distribution.

Note: If the `grubby` command does not update the kernels of a RHEL 7.x or CentOS 7.x system, you can manually update all kernels on the system. For example, to add the parameter `transparent_hugepage=never` to all kernels on a system.

1. Add the parameter to the `GRUB_CMDLINE_LINUX` line in the file parameter in `/etc/default/grub`.

   ```
   GRUB_TIMEOUT=5
   GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
   GRUB_DEFAULT=saved
   GRUB_DISABLE_SUBMENU=true
   GRUB_TERMINAL_OUTPUT="console"
   GRUB_CMDLINE_LINUX="crashkernel=auto rd.lvm.lv=cl/root rd.lvm.lv=cl/
   swap rhgb quiet transparent_hugepage=never"
   GRUB_DISABLE_RECOVERY="true"
   ```

2. As root, run the `grub2-mkconfig` command to update the kernels.

   ```
   # grub2-mkconfig -o /boot/grub2/grub.cfg
   ```

3. Reboot the system.

Synchronizing System Clocks

You should use NTP (Network Time Protocol) to synchronize the system clocks on all hosts that comprise your Greenplum Database system. See *www.ntp.org* for more information about NTP.

NTP on the segment hosts should be configured to use the master host as the primary time source, and the standby master as the secondary time source. On the master and standby master hosts, configure NTP to point to your preferred time server.

To configure NTP

1. On the master host, log in as root and edit the `/etc/ntp.conf` file. Set the `server` parameter to point to your data center's NTP time server. For example (if `10.6.220.20` was the IP address of your data center's NTP server):

   ```
   server 10.6.220.20
   ```

2. On each segment host, log in as root and edit the `/etc/ntp.conf` file. Set the first `server` parameter to point to the master host, and the second server parameter to point to the standby master host. For example:

   ```
   server mdw prefer
   server smdw
   ```

3. On the standby master host, log in as root and edit the `/etc/ntp.conf` file. Set the first `server` parameter to point to the primary master host, and the second server parameter to point to your data center's NTP time server. For example:

   ```
   server mdw prefer
   server 10.6.220.20
   ```

4. On the master host, use the NTP daemon synchronize the system clocks on all Greenplum hosts. For example, using *gpssh*:

   ```
   # gpssh -f hostfile_gpssh_allhosts -v -e 'ntpd'
   ```

Creating the Greenplum Administrative User

Create a dedicated operating system user account on each node to run and administer Greenplum Database. This user account is named `gpadmin` by convention.

> **Important:** You cannot run the Greenplum Database server as `root`.

The `gpadmin` user must have permission to access the services and directories required to install and run Greenplum Database.

The `gpadmin` user on each Greenplum host must have an SSH key pair installed and be able to SSH from any host in the cluster to any other host in the cluster without entering a password or passphrase (called "passwordless SSH"). If you enable passwordless SSH from the master host to every other host in the cluster ("1-*n* passwordless SSH"), you can use the Greenplum Database `gpssh-exkeys` command-line utility later to enable passwordless SSH from every host to every other host ("*n*-*n* passwordless SSH").

You can optionally give the `gpadmin` user sudo privilege, so that you can easily administer all hosts in the Greenplum Database cluster as `gpadmin` using the `sudo`, `ssh/scp`, and `gpssh/gpscp` commands.

The following steps show how to set up the `gpadmin` user on a host, set a password, create an SSH key pair, and (optionally) enable sudo capability. These steps must be performed as root on every Greenplum Database cluster host. (For a large Greenplum Database cluster you will want to automate these steps using your system provisioning tools.)

> **Note:** See *Example Ansible Playbook* for an example that shows how to automate the tasks of creating the `gpadmin` user and installing the Greenplum Database software on all hosts in the cluster.

1. Create the `gpadmin` group and user.

   > **Note:** If you are installing Greenplum Database on RHEL 7.2 or CentOS 7.2 and want to disable IPC object removal by creating the `gpadmin` user as a system account, provide both the `-r` option (create the user as a system account) and the `-m` option (create a home directory) to the `useradd` command. On Ubuntu systems, you must use the `-m` option with the `useradd` command to create a home directory for a user.

This example creates the `gpadmin` group, creates the `gpadmin` user as a system account with a home directory and as a member of the `gpadmin` group, and creates a password for the user.

```
# groupadd gpadmin
# useradd gpadmin -r -m -g gpadmin
# passwd gpadmin
New password: <changeme>
Retype new password: <changeme>
```

> **Note:** Make sure the `gpadmin` user has the same user id (uid) and group id (gid) numbers on each host to prevent problems with scripts or services that use them for identity or permissions. For example, backing up Greenplum databases to some networked filesy stems or storage appliances could fail if the `gpadmin` user has different uid or gid numbers on different segment hosts. When you create the `gpadmin` group and user, you can use the `groupadd -g` option to specify a gid number and the `useradd -u` option to specify the uid number. Use the command `id gpadmin` to see the uid and gid for the `gpadmin` user on the current host.

2. Switch to the `gpadmin` user and generate an SSH key pair for the `gpadmin` user.

```
$ su gpadmin
$ ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Enter file in which to save the key (/home/gpadmin/.ssh/id_rsa):
Created directory '/home/gpadmin/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

At the passphrase prompts, press Enter so that SSH connections will not require entry of a passphrase.

3. (Optional) Grant sudo access to the `gpadmin` user.

On Red Hat or CentOS, run `visudo` and uncomment the `%wheel` group entry.

```
%wheel        ALL=(ALL)       NOPASSWD: ALL
```

Make sure you uncomment the line that has the `NOPASSWD` keyword.

Add the `gpadmin` user to the `wheel` group with this command.

```
# usermod -aG wheel gpadmin
```

Next Steps

- *Installing Greenplum Database Software*
- *Validating Your Systems*
- *Initializing a Greenplum Database System*

Installing the Greenplum Database Software

Describes how to install the Greenplum Database software binaries on all of the hosts that will comprise your Greenplum Database system, how to enable passwordless SSH for the `gpadmin` user, and how to verify the installation.

Perform the following tasks in order:

1. *Installing Greenplum Database*
2. *Enabling Passwordless SSH*
3. *Confirm the software installation.*
4. *Next Steps*

Installing Greenplum Database

You must install Greenplum Database on each host machine of the Greenplum Database system. Pivotal distributes the Greenplum Database software as a downloadable package that you install on each host system with the operating system's package management system. You can download the package from *Pivotal Network*.

Before you begin installing Greenplum Database, be sure you have completed the steps in *Configuring Your Systems* to configure each of the master, standby master, and segment host machines for Greenplum Database.

> **Important:** After installing Greenplum Database, you must set Greenplum Database environment variables. See *Setting Greenplum Environment Variables*.

See *Example Ansible Playbook* for an example script that shows how you can automate creating the `gpadmin` user and installing the Greenplum Database.

Follow these instructions to install Greenplum Database.

> **Important:** You require sudo or root user access to install from a pre-built RPM or DEB file.

1. Download and copy the Greenplum Database package to the `gpadmin` user's home directory on the master, standby master, and every segment host machine. The distribution file name has the format `greenplum-db-<version>-<platform>.rpm` for RHEL, CentOS, and Oracle Linux systems, or `greenplum-db-<version>-<platform>.deb` for Ubuntu systems, where `<platform>` is similar to `rhel7-x86_64` (Red Hat 7 64-bit).

   > **Note:** For Oracle Linux installations, download and install the `rhel7-x86_64`distribution files.
2. With sudo (or as root), install the Greenplum Database package on each host machine using your system's package manager software.

   - For RHEL/CentOS systems, execute the `yum` command:

     ```
     $ sudo yum install ./greenplum-db-<version>-<platform>.rpm
     ```

   - For Ubuntu systems, execute the `apt` command:

     ```
     $ sudo apt install ./greenplum-db-<version>-<platform>.deb
     ```

   The `yum` or `apt` command automatically installs software dependencies, copies the Greenplum Database software files into a version-specific directory under `/usr/local`, `/usr/local/greenplum-db-<version>`, and creates the symbolic link `/usr/local/greenplum-db` to the installation directory.
3. Change the owner and group of the installed files to `gpadmin`:

   ```
   $ sudo chown -R gpadmin:gpadmin /usr/local/greenplum*
   ```

```
$ sudo chgrp -R gpadmin /usr/local/greenplum*
```

(Optional) Installing to a Non-Default Directory

On RHEL/CentOS systems, you can use the `rpm` command with the `--prefix` option to install Greenplum Database to a non-default directory (instead of under `/usr/local`). Note, however, that using `rpm` does not automatically install Greenplum Database dependencies; you must manually install dependencies to each host system.

Follow these instructions to install Greenplum Database to a specific directory.

Important: You require sudo or root user access to install from a pre-built RPM file.

1. Download and copy the Greenplum Database package to the `gpadmin` user's home directory on the master, standby master, and every segment host machine. The distribution file name has the format `greenplum-db-<version>-<platform>.rpm` for RHEL and CentOS systems, or `greenplum-db-<version>-<platform>.deb` for Ubuntu systems, where `<platform>` is similar to `rhel7-x86_64` (Red Hat 7 64-bit).
2. Manually install the Greenplum Database dependencies to each host system. For RHEL/CentOS 7:

   ```
   $ sudo yum install apr apr-util bash bzip2 curl krb5 libcurl libevent \
   libxml2 libyaml zlib openldap openssh openssl openssl-libs perl readline
    rsync R sed tar zip
   ```

   For RHEL/CentOS 6:

   ```
   $ sudo yum install apr apr-util bash bzip2 curl krb5 libcurl libevent2 \
   libxml2 libyaml zlib openldap openssh openssl openssl-libs perl readline
    rsync R sed tar zip
   ```

3. Use `rpm` with the `--prefix` option to install the Greenplum Database package to your chosen installation directory on each host machine:

   ```
   $ sudo rpm --install ./greenplum-db-<version>-<platform>.rpm --
   prefix=<directory>
   ```

   The `rpm` command copies the Greenplum Database software files into a version-specific directory under your chosen `<directory>`, `<directory>/greenplum-db-<version>`, and creates the symbolic link `<directory>/greenplum-db` to the versioned directory.
4. Change the owner and group of the installed files to `gpadmin`:

   ```
   $ sudo chown -R gpadmin:gpadmin <directory>/greenplum*
   ```

   Note: All example procedures in the Greenplum Database documentation assume that you installed to the default directory, `/usr/local`. If you install to a non-default directory, substitute that directory for `/usr/local`.

   If you install to a non-default directory using `rpm`, you will need to continue using `rpm` (and of `yum`) to perform minor version upgrades; these changes are covered in the upgrade documentation.

Enabling Passwordless SSH

The `gpadmin` user on each Greenplum host must be able to SSH from any host in the cluster to any other host in the cluster without entering a password or passphrase (called "passwordless SSH"). If you enable passwordless SSH from the master host to every other host in the cluster ("1-*n* passwordless SSH"), you can use the Greenplum Database `gpssh-exkeys` command-line utility to enable passwordless SSH from every host to every other host ("*n*-*n* passwordless SSH").

1. Log in to the master host as the `gpadmin` user.

2. Source the `path` file in the Greenplum Database installation directory.

```
$ source /usr/local/greenplum-db-<version>/greenplum_path.sh
```

   Note: Add the above `source` command to the `gpadmin` user's `.bashrc` or other shell startup file so that the Greenplum Database path and environment variables are set whenever you log in as `gpadmin`.

3. Use the `ssh-copy-id` command to add the `gpadmin` user's public key to the `authorized_hosts` SSH file on every other host in the cluster.

```
$ ssh-copy-id smdw
$ ssh-copy-id sdw1
$ ssh-copy-id sdw2
$ ssh-copy-id sdw3
. . .
```

   This enables 1-*n* passwordless SSH. You will be prompted to enter the `gpadmin` user's password for each host. If you have the `sshpass` command on your system, you can use a command like the following to avoid the prompt.

```
$ SSHPASS=<password> sshpass -e ssh-copy-id smdw
```

4. In the `gpadmin` home directory, create a file named `hostfile_exkeys` that has the machine configured host names and host addresses (interface names) for each host in your Greenplum system (master, standby master, and segment hosts). Make sure there are no blank lines or extra spaces. Check the `/etc/hosts` file on your systems for the correct host names to use for your environment. For example, if you have a master, standby master, and three segment hosts with two unbonded network interfaces per host, your file would look something like this:

```
mdw
mdw-1
mdw-2
smdw
smdw-1
smdw-2
sdw1
sdw1-1
sdw1-2
sdw2
sdw2-1
sdw2-2
sdw3
sdw3-1
sdw3-2
```

5. Run the `gpssh-exkeys` utility with your `hostfile_exkeys` file to enable *n-n* passwordless SSH for the `gpadmin` user.

```
$ gpssh-exkeys -f hostfile_exkeys
```

Confirming Your Installation

To make sure the Greenplum software was installed and configured correctly, run the following confirmation steps from your Greenplum master host. If necessary, correct any problems before continuing on to the next task.

1. Log in to the master host as `gpadmin`:

```
$ su - gpadmin
```

2. Use the `gpssh` utility to see if you can log in to all hosts without a password prompt, and to confirm that the Greenplum software was installed on all hosts. Use the `hostfile_exkeys` file you used to set up passwordless SSH. For example:

   ```
   $ gpssh -f hostfile_exkeys -e 'ls -l /usr/local/greenplum-db-<version>'
   ```

   If the installation was successful, you should be able to log in to all hosts without a password prompt. All hosts should show that they have the same contents in their installation directories, and that the directories are owned by the `gpadmin` user.

   If you are prompted for a password, run the following command to redo the ssh key exchange:

   ```
   $ gpssh-exkeys -f hostfile_exkeys
   ```

About Your Greenplum Database Installation

- `greenplum_path.sh` — This file contains the environment variables for Greenplum Database. See *Setting Greenplum Environment Variables*.
- **bin** — This directory contains the Greenplum Database management utilities. This directory also contains the PostgreSQL client and server programs, most of which are also used in Greenplum Database.
- **docs/cli_help** — This directory contains help files for Greenplum Database command-line utilities.
- **docs/cli_help/gpconfigs** — This directory contains sample `gpinitsystem` configuration files and host files that can be modified and used when installing and initializing a Greenplum Database system.
- **ext** — Bundled programs (such as Python) used by some Greenplum Database utilities.
- **include** — The C header files for Greenplum Database.
- **lib** — Greenplum Database and PostgreSQL library files.
- **sbin** — Supporting/Internal scripts and programs.
- **share** — Shared files for Greenplum Database.

Next Steps

- *Creating the Data Storage Areas*
- *Validating Your Systems*
- *Initializing a Greenplum Database System*

Creating the Data Storage Areas

Describes how to create the directory locations where Greenplum Database data is stored for each master, standby, and segment instance.

Creating Data Storage Areas on the Master and Standby Master Hosts

A data storage area is required on the Greenplum Database master and standby master hosts to store Greenplum Database system data such as catalog data and other system metadata.

To create the data directory location on the master

The data directory location on the master is different than those on the segments. The master does not store any user data, only the system catalog tables and system metadata are stored on the master instance, therefore you do not need to designate as much storage space as on the segments.

1. Create or choose a directory that will serve as your master data storage area. This directory should have sufficient disk space for your data and be owned by the `gpadmin` user and group. For example, run the following commands as `root`:

```
# mkdir -p /data/master
```

2. Change ownership of this directory to the `gpadmin` user. For example:

```
# chown gpadmin:gpadmin /data/master
```

3. Using *gpssh*, create the master data directory location on your standby master as well. For example:

```
# source /usr/local/greenplum-db/greenplum_path.sh
# gpssh -h smdw -e 'mkdir -p /data/master'
# gpssh -h smdw -e 'chown gpadmin:gpadmin /data/master'
```

Creating Data Storage Areas on Segment Hosts

Data storage areas are required on the Greenplum Database segment hosts for primary segments. Separate storage areas are required for mirror segments.

To create the data directory locations on all segment hosts

1. On the master host, log in as `root`:

```
# su
```

2. Create a file called `hostfile_gpssh_segonly`. This file should have only one machine configured host name for each segment host. For example, if you have three segment hosts:

```
sdw1
sdw2
sdw3
```

3. Using *gpssh*, create the primary and mirror data directory locations on all segment hosts at once using the `hostfile_gpssh_segonly` file you just created. For example:

```
# source /usr/local/greenplum-db/greenplum_path.sh
# gpssh -f hostfile_gpssh_segonly -e 'mkdir -p /data/primary'
```

```
# gpssh -f hostfile_gpssh_segonly -e 'mkdir -p /data/mirror'
# gpssh -f hostfile_gpssh_segonly -e 'chown -R gpadmin /data/*'
```

Next Steps

- *Validating Your Systems*
- *Initializing a Greenplum Database System*

Validating Your Systems

Validate your hardware and network performance.

Greenplum provides a management utility called *gpcheckperf*, which can be used to identify hardware and system-level issues on the machines in your Greenplum Database array. *gpcheckperf* starts a session on the specified hosts and runs the following performance tests:

- Network Performance (`gpnetbench*`)
- Disk I/O Performance (`dd` test)
- Memory Bandwidth (`stream` test)

Before using `gpcheckperf`, you must have a trusted host setup between the hosts involved in the performance test. You can use the utility *gpssh-exkeys* to update the known host files and exchange public keys between hosts if you have not done so already. Note that `gpcheckperf` calls to *gpssh* and *gpscp*, so these Greenplum utilities must be in your `$PATH`.

Validating Network Performance

To test network performance, run *gpcheckperf* with one of the network test run options: parallel pair test (`-r N`), serial pair test (`-r n`), or full matrix test (`-r M`). The utility runs a network benchmark program that transfers a 5 second stream of data from the current host to each remote host included in the test. By default, the data is transferred in parallel to each remote host and the minimum, maximum, average and median network transfer rates are reported in megabytes (MB) per second. If the summary transfer rate is slower than expected (less than 100 MB/s), you can run the network test serially using the `-r n` option to obtain per-host results. To run a full-matrix bandwidth test, you can specify `-r M` which will cause every host to send and receive data from every other host specified. This test is best used to validate if the switch fabric can tolerate a full-matrix workload.

Most systems in a Greenplum Database array are configured with multiple network interface cards (NICs), each NIC on its own subnet. When testing network performance, it is important to test each subnet individually. For example, considering the following network configuration of two NICs per host:

Table 6: Example Network Interface Configuration

Greenplum Host	Subnet1 NICs	Subnet2 NICs
Segment 1	sdw1-1	sdw1-2
Segment 2	sdw2-1	sdw2-2
Segment 3	sdw3-1	sdw3-2

You would create four distinct host files for use with the *gpcheckperf* network test:

Table 7: Example Network Test Host File Contents

hostfile_gpchecknet_ic1	hostfile_gpchecknet_ic2
sdw1-1	sdw1-2
sdw2-1	sdw2-2
sdw3-1	sdw3-2

You would then run *gpcheckperf* once per subnet. For example (if testing an *even* number of hosts, run in parallel pairs test mode):

```
$ gpcheckperf -f hostfile_gpchecknet_ic1 -r N -d /tmp > subnet1.out
$ gpcheckperf -f hostfile_gpchecknet_ic2 -r N -d /tmp > subnet2.out
```

If you have an *odd* number of hosts to test, you can run in serial test mode (`-r n`).

Validating Disk I/O and Memory Bandwidth

To test disk and memory bandwidth performance, run *gpcheckperf* with the disk and stream test run options (`-r ds`). The disk test uses the `dd` command (a standard UNIX utility) to test the sequential throughput performance of a logical disk or file system. The memory test uses the STREAM benchmark program to measure sustainable memory bandwidth. Results are reported in MB per second (MB/s).

To run the disk and stream tests

1. Log in on the master host as the `gpadmin` user.
2. Source the `greenplum_path.sh` path file from your Greenplum installation. For example:

   ```
   $ source /usr/local/greenplum-db/greenplum_path.sh
   ```

3. Create a host file named `hostfile_gpcheckperf` that has one host name per segment host. Do not include the master host. For example:

   ```
   sdw1
   sdw2
   sdw3
   sdw4
   ```

4. Run the `gpcheckperf` utility using the `hostfile_gpcheckperf` file you just created. Use the `-d` option to specify the file systems you want to test on each host (you must have write access to these directories). You will want to test all primary and mirror segment data directory locations. For example:

   ```
   $ gpcheckperf -f hostfile_gpcheckperf -r ds -D \
     -d /data1/primary -d  /data2/primary \
     -d /data1/mirror -d  /data2/mirror
   ```

5. The utility may take a while to perform the tests as it is copying very large files between the hosts. When it is finished you will see the summary results for the Disk Write, Disk Read, and Stream tests.

Initializing a Greenplum Database System

Describes how to initialize a Greenplum Database database system.

The instructions in this chapter assume you have already prepared your hosts as described in *Configuring Your Systems* and installed the Greenplum Database software on all of the hosts in the system according to the instructions in *Installing the Greenplum Database Software*.

This chapter contains the following topics:

- *Overview*
- *Initializing Greenplum Database*
- *Setting Greenplum Environment Variables*
- *Next Steps*

Overview

Because Greenplum Database is distributed, the process for initializing a Greenplum Database management system (DBMS) involves initializing several individual PostgreSQL database instances (called *segment instances* in Greenplum).

Each database instance (the master and all segments) must be initialized across all of the hosts in the system in such a way that they can all work together as a unified DBMS. Greenplum provides its own version of `initdb` called *`gpinitsystem`*, which takes care of initializing the database on the master and on each segment instance, and starting each instance in the correct order.

After the Greenplum Database database system has been initialized and started, you can then create and manage databases as you would in a regular PostgreSQL DBMS by connecting to the Greenplum master.

Initializing Greenplum Database

These are the high-level tasks for initializing Greenplum Database:

1. Make sure you have completed all of the installation tasks described in *Configuring Your Systems* and *Installing the Greenplum Database Software*.
2. Create a host file that contains the host addresses of your *segments*. See *Creating the Initialization Host File*.
3. Create your Greenplum Database system configuration file. See *Creating the Greenplum Database Configuration File*.
4. By default, Greenplum Database will be initialized using the locale of the master host system. Make sure this is the correct locale you want to use, as some locale options cannot be changed after initialization. See *Configuring Timezone and Localization Settings* for more information.
5. Run the Greenplum Database initialization utility on the master host. See *Running the Initialization Utility*.
6. Set the Greenplum Database timezone. See *Setting the Greenplum Database Timezone*.
7. Set environment variables for the Greenplum Database user. See *Setting Greenplum Environment Variables*.

When performing the following initialization tasks, you must be logged into the master host as the `gpadmin` user, and to run Greenplum Database utilities, you must source the `greenplum_path.sh` file to set Greenplum Database environment variables. For example, if you are logged into the master, run these commands.

```
$ su - gpadmin
$ source /usr/local/greenplum-db/greenplum_path.sh
```

Creating the Initialization Host File

The *gpinitsystem* utility requires a host file that contains the list of addresses for each segment host. The initialization utility determines the number of segment instances per host by the number host addresses listed per host times the number of data directory locations specified in the `gpinitsystem_config` file.

This file should only contain *segment* host addresses (not the master or standby master). For segment machines with multiple, unbonded network interfaces, this file should list the host address names for each interface — one per line.

> **Note:** The Greenplum Database segment host naming convention is *sdwN* where *sdw* is a prefix and *N* is an integer. For example, `sdw2` and so on. If hosts have multiple unbonded NICs, the convention is to append a dash (`-`) and number to the host name. For example, `sdw1-1` and `sdw1-2` are the two interface names for host `sdw1`. However, NIC bonding is recommended to create a load-balanced, fault-tolerant network.

To create the initialization host file

1. Create a file named `hostfile_gpinitsystem`. In this file add the host address name(s) of your *segment* host interfaces, one name per line, no extra lines or spaces. For example, if you have four segment hosts with two unbonded network interfaces each:

```
sdw1-1
sdw1-2
sdw2-1
sdw2-2
sdw3-1
sdw3-2
sdw4-1
sdw4-2
```

2. Save and close the file.

> **Note:** If you are not sure of the host names and/or interface address names used by your machines, look in the `/etc/hosts` file.

Creating the Greenplum Database Configuration File

Your Greenplum Database configuration file tells the *gpinitsystem* utility how you want to configure your Greenplum Database system. An example configuration file can be found in `$GPHOME/docs/cli_help/gpconfigs/gpinitsystem_config`.

To create a gpinitsystem_config file

1. Make a copy of the `gpinitsystem_config` file to use as a starting point. For example:

```
$ cp $GPHOME/docs/cli_help/gpconfigs/gpinitsystem_config \
     /home/gpadmin/gpconfigs/gpinitsystem_config
```

2. Open the file you just copied in a text editor.

   Set all of the required parameters according to your environment. See *gpinitsystem* for more information. A Greenplum Database system must contain a master instance and at *least two* segment instances (even if setting up a single node system).

   The `DATA_DIRECTORY` parameter is what determines how many segments per host will be created. If your segment hosts have multiple network interfaces, and you used their interface address names in your host file, the number of segments will be evenly spread over the number of available interfaces.

To specify PORT_BASE, review the port range specified in the net.ipv4.ip_local_port_range parameter in the /etc/sysctl.conf file. See *Recommended OS Parameters Settings*.

Here is an example of the *required* parameters in the gpinitsystem_config file:

```
ARRAY_NAME="Greenplum Data Platform"
SEG_PREFIX=gpseg
PORT_BASE=6000
declare -a DATA_DIRECTORY=(/data1/primary /data1/primary /data1/primary /
data2/primary /data2/primary /data2/primary)
MASTER_HOSTNAME=mdw
MASTER_DIRECTORY=/data/master
MASTER_PORT=5432
TRUSTED SHELL=ssh
CHECK_POINT_SEGMENTS=8
ENCODING=UNICODE
```

3. (Optional) If you want to deploy mirror segments, uncomment and set the mirroring parameters according to your environment. To specify MIRROR_PORT_BASE, review the port range specified under the net.ipv4.ip_local_port_range parameter in the /etc/sysctl.conf file. Here is an example of the *optional* mirror parameters in the gpinitsystem_config file:

```
MIRROR_PORT_BASE=7000
declare -a MIRROR_DATA_DIRECTORY=(/data1/mirror /data1/mirror /data1/
mirror /data2/mirror /data2/mirror /data2/mirror)
```

   Note: You can initialize your Greenplum system with primary segments only and deploy mirrors later using the *gpaddmirrors* utility.

4. Save and close the file.

Running the Initialization Utility

The *gpinitsystem* utility will create a Greenplum Database system using the values defined in the configuration file.

These steps assume you are logged in as the gpadmin user and have sourced the greenplum_path.sh file to set Greenplum Database environment variables.

To run the initialization utility

1. Run the following command referencing the path and file name of your initialization configuration file (gpinitsystem_config) and host file (hostfile_gpinitsystem). For example:

```
$ cd ~
$ gpinitsystem -c gpconfigs/gpinitsystem_config -h gpconfigs/
hostfile_gpinitsystem
```

   For a fully redundant system (with a standby master and a *spread* mirror configuration) include the -s and -S options. For example:

```
$ gpinitsystem -c gpconfigs/gpinitsystem_config -h gpconfigs/
hostfile_gpinitsystem \
  -s standby_master_hostname -S
```

   During a new cluster creation, you may use the *-O output_configuration_file* option to save the cluster configuration details in a file. For example:

```
$ gpinitsystem -c gpconfigs/gpinitsystem_config -O gpconfigs/
config_template
```

This output file can be edited and used at a later stage as the input file of the `-I` option, to create a new cluster or to recover from a backup. See *gpinitsystem* for further details.

2. The utility will verify your setup information and make sure it can connect to each host and access the data directories specified in your configuration. If all of the pre-checks are successful, the utility will prompt you to confirm your configuration. For example:

```
=> Continue with Greenplum creation? Yy/Nn
```

3. Press y to start the initialization.
4. The utility will then begin setup and initialization of the master instance and each segment instance in the system. Each segment instance is set up in parallel. Depending on the number of segments, this process can take a while.
5. At the end of a successful setup, the utility will start your Greenplum Database system. You should see:

```
=> Greenplum Database instance successfully created.
```

Troubleshooting Initialization Problems

If the utility encounters any errors while setting up an instance, the entire process will fail, and could possibly leave you with a partially created system. Refer to the error messages and logs to determine the cause of the failure and where in the process the failure occurred. Log files are created in `~/gpAdminLogs`.

Depending on when the error occurred in the process, you may need to clean up and then try the `gpinitsystem` utility again. For example, if some segment instances were created and some failed, you may need to stop `postgres` processes and remove any utility-created data directories from your data storage area(s). A backout script is created to help with this cleanup if necessary.

Using the Backout Script

If the gpinitsystem utility fails, it will create the following backout script if it has left your system in a partially installed state:

`~/gpAdminLogs/backout_gpinitsystem_<user>_<timestamp>`

You can use this script to clean up a partially created Greenplum Database system. This backout script will remove any utility-created data directories, `postgres` processes, and log files. After correcting the error that caused `gpinitsystem` to fail and running the backout script, you should be ready to retry initializing your Greenplum Database array.

The following example shows how to run the backout script:

```
$ sh backout_gpinitsystem_gpadmin_20071031_121053
```

Setting the Greenplum Database Timezone

As a best practice, configure Greenplum Database and the host systems to use a known, supported timezone. Greenplum Database uses a timezone from a set of internally stored PostgreSQL timezones. Setting the Greenplum Database timezone prevents Greenplum Database from selecting a timezone each time the cluster is restarted and sets the timezone for the Greenplum Database master and segment instances.

Use the *gpconfig* utility to show and set the Greenplum Database timezone. For example, these commands show the Greenplum Database timezone and set the timezone to `US/Pacific`.

```
$ gpconfig -s TimeZone
$ gpconfig -c TimeZone -v 'US/Pacific'
```

You must restart Greenplum Database after changing the timezone. The command `gpstop -ra` restarts Greenplum Database. The catalog view `pg_timezone_names` provides Greenplum Database timezone information.

For more information about the Greenplum Database timezone, see *Configuring Timezone and Localization Settings*.

Setting Greenplum Environment Variables

You must set environment variables in the Greenplum Database user (`gpadmin`) environment that runs Greenplum Database on the Greenplum Database master and standby master hosts. A `greenplum_path.sh` file is provided in the Greenplum Database installation directory with environment variable settings for Greenplum Database.

The Greenplum Database management utilities also require that the `MASTER_DATA_DIRECTORY` environment variable be set. This should point to the directory created by the `gpinitsystem` utility in the master data directory location.

> **Note:** The `greenplum_path.sh` script changes the operating environment in order to support running the Greenplum Database-specific utilities. These same changes to the environment can negatively affect the operation of other system-level utilities, such as `ps` or `yum`. Use separate accounts for performing system administration and database administration, instead of attempting to perform both functions as `gpadmin`.

These steps ensure that the environment variables are set for the `gpadmin` user after a system reboot.

To set up the gpadmin environment for Greenplum Database

1. Open the `gpadmin` profile file (such as `.bashrc`) in a text editor. For example:

```
$ vi ~/.bashrc
```

2. Add lines to this file to source the `greenplum_path.sh` file and set the `MASTER_DATA_DIRECTORY` environment variable. For example:

```
source /usr/local/greenplum-db/greenplum_path.sh
export MASTER_DATA_DIRECTORY=/data/master/gpseg-1
```

3. (Optional) You may also want to set some client session environment variables such as `PGPORT`, `PGUSER` and `PGDATABASE` for convenience. For example:

```
export PGPORT=5432
export PGUSER=gpadmin
export PGDATABASE=default_login_database_name
```

4. (Optional) If you use RHEL 7 or CentOS 7, add the following line to the end of the `.bashrc` file to enable using the `ps` command in the `greenplum_path.sh` environment:

```
export LD_PRELOAD=/lib64/libz.so.1 ps
```

5. Save and close the file.
6. After editing the profile file, source it to make the changes active. For example:

```
$ source ~/.bashrc
```

7. If you have a standby master host, copy your environment file to the standby master as well. For example:

```
$ cd ~
$ scp .bashrc standby_hostname:`pwd`
```

Note: The `.bashrc` file should not produce any output. If you wish to have a message display to users upon logging in, use the `.bash_profile` file instead.

Next Steps

After your system is up and running, the next steps are:

- *Allowing Client Connections*
- *Creating Databases and Loading Data*

Allowing Client Connections

After a Greenplum Database is first initialized it will only allow local connections to the database from the `gpadmin` role (or whatever system user ran `gpinitsystem`). If you would like other users or client machines to be able to connect to Greenplum Database, you must give them access. See the *Greenplum Database Administrator Guide* for more information.

Creating Databases and Loading Data

After verifying your installation, you may want to begin creating databases and loading data. See *Defining Database Objects* and *Loading and Unloading Data* in the *Greenplum Database Administrator Guide* for more information about creating databases, schemas, tables, and other database objects in Greenplum Database and loading your data.

Installing Optional Extensions

Information about installing optional Greenplum Database extensions and packages, such as the Procedural Language extensions and the Python and R Data Science Packages.

Procedural Language, Machine Learning, and Geospatial Extensions

Optional. Use the Greenplum package manager (`gppkg`) to install Greenplum Database extensions such as PL/Java, PL/R, PostGIS, and MADlib, along with their dependencies, across an entire cluster. The package manager also integrates with existing scripts so that any packages are automatically installed on any new hosts introduced into the system following cluster expansion or segment host recovery.

See *gppkg* for more information, including usage.

Extension packages can be downloaded from the Greenplum Database page on *Pivotal Network*. The extension documentation in the *Greenplum Database Reference Guide* contains information about installing extension packages and using extensions.

- *Greenplum PL/R Language Extension*
- *Greenplum PL/Java Language Extension*
- *Greenplum MADlib Extension for Analytics*
- *Greenplum PostGIS Extension*

> **Important:** If you intend to use an extension package with Greenplum Database 6 you must install and use a Greenplum Database extension package (gppkg files and contrib modules) that is built for Greenplum Database 6. Any custom modules that were used with earlier versions must be rebuilt for use with Greenplum Database 6.

Python Data Science Module Package

Greenplum Database provides a collection of data science-related Python modules that can be used with the Greenplum Database PL/Python language. You can download these modules in `.gppkg` format from *Pivotal Network*.

This section contains the following information:

- *Python Data Science Modules*
- *Installing the Python Data Science Module Package*
- *Uninstalling the Python Data Science Module Package*

For information about the Greenplum Database PL/Python Language, see *Greenplum PL/Python Language Extension*.

Python Data Science Modules

Modules provided in the Python Data Science package include:

Table 8: Data Science Modules

Module Name	Description/Used For
atomicwrites	Atomic file writes
attrs	Declarative approach for defining class attributes
Autograd	Gradient-based optimization

Module Name	Description/Used For
backports.functools-lru-cache	Backports `functools.lru_cache` from Python 3.3
Beautiful Soup	Navigating HTML and XML
Blis	Blis linear algebra routines
Boto	Amazon Web Services library
Boto3	The AWS SDK
botocore	Low-level, data-driven core of boto3
Bottleneck	Fast NumPy array functions
Bz2file	Read and write bzip2-compressed files
Certifi	Provides Mozilla CA bundle
Chardet	Universal encoding detector for Python 2 and 3
ConfigParser	Updated `configparser` module
contextlib2	Backports and enhancements for the `contextlib` module
Cycler	Composable style cycles
cymem	Manage calls to calloc/free through Cython
Docutils	Python documentation utilities
enum34	Backport of Python 3.4 Enum
Funcsigs	Python function signatures from PEP362
functools32	Backport of the `functools` module from Python 3.2.3
funcy	Functional tools focused on practicality
future	Compatibility layer between Python 2 and Python 3
futures	Backport of the `concurrent.futures` package from Python 3
Gensim	Topic modeling and document indexing
GluonTS (Python 3 only)	Probabilistic time series modeling
h5py	Read and write HDF5 files
idna	Internationalized Domain Names in Applications (IDNA)
importlib-metadata	Read metadata from Python packages
Jinja2	Stand-alone template engine
JMESPath	JSON Matching Expressions
Joblib	Python functions as pipeline jobs
jsonschema	JSON Schema validation
Keras (RHEL/CentOS 7 only)	Deep learning
Keras Applications	Reference implementations of popular deep learning models
Keras Preprocessing	Easy data preprocessing and data augmentation for deep learning models
Kiwi	A fast implementation of the Cassowary constraint solver

Module Name	Description/Used For
Lifelines	Survival analysis
lxml	XML and HTML processing
MarkupSafe	Safely add untrusted strings to HTML/XML markup
Matplotlib	Python plotting package
mock	Rolling backport of `unittest.mock`
more-itertools	More routines for operating on iterables, beyond itertools
MurmurHash	Cython bindings for MurmurHash
NLTK	Natural language toolkit
NumExpr	Fast numerical expression evaluator for NumPy
NumPy	Scientific computing
packaging	Core utilities for Python packages
Pandas	Data analysis
pathlib, pathlib2	Object-oriented filesystem paths
patsy	Package for describing statistical models and for building design matrices
Pattern-en	Part-of-speech tagging
pip	Tool for installing Python packages
plac	Command line arguments parser
pluggy	Plugin and hook calling mechanisms
preshed	Cython hash table that trusts the keys are pre-hashed
protobuf	Protocol buffers
py	Cross-python path, ini-parsing, io, code, log facilities
pyLDAvis	Interactive topic model visualization
PyMC3	Statistical modeling and probabilistic machine learning
pyparsing	Python parsing
pytest	Testing framework
python-dateutil	Extensions to the standard Python datetime module
pytz	World timezone definitions, modern and historical
PyYAML	YAML parser and emitter
requests	HTTP library
s3transfer	Amazon S3 transfer manager
scandir	Directory iteration function
scikit-learn	Machine learning data mining and analysis
SciPy	Scientific computing
setuptools	Download, build, install, upgrade, and uninstall Python packages

Module Name	Description/Used For
six	Python 2 and 3 compatibility library
smart-open	Utilities for streaming large files (S3, HDFS, gzip, bz2, and so forth)
spaCy	Large scale natural language processing
srsly	Modern high-performance serialization utilities for Python
StatsModels	Statistical modeling
subprocess32	Backport of the subprocess module from Python 3
Tensorflow (RHEL/CentOS 7 only)	Numerical computation using data flow graphs
Theano	Optimizing compiler for evaluating mathematical expressions on CPUs and GPUs
thinc	Practical Machine Learning for NLP
tqdm	Fast, extensible progress meter
urllib3	HTTP library with thread-safe connection pooling, file post, and more
wasabi	Lightweight console printing and formatting toolkit
wcwidth	Measures number of Terminal column cells of wide-character codes
Werkzeug	Comprehensive WSGI web application library
wheel	A built-package format for Python
XGBoost	Gradient boosting, classifying, ranking
zipp	Backport of pathlib-compatible object wrapper for zip files

Installing the Python Data Science Module Package

Before you install the Python Data Science Module package, make sure that your Greenplum Database is running, you have sourced `greenplum_path.sh`, and that the `$MASTER_DATA_DIRECTORY` and `$GPHOME` environment variables are set.

> **Note:** The `PyMC3` module depends on `Tk`. If you want to use `PyMC3`, you must install the `tk` OS package on every node in your cluster. For example:
>
> ```
> $ yum install tk
> ```

1. Locate the Python Data Science module package that you built or downloaded.

   The file name format of the package is `DataSciencePython-<version>-relhel<N>-x86_64.gppkg`.
2. Copy the package to the Greenplum Database master host.
3. Use the `gppkg` command to install the package. For example:

   ```
   $ gppkg -i DataSciencePython-<version>-relhel<N>_x86_64.gppkg
   ```

   `gppkg` installs the Python Data Science modules on all nodes in your Greenplum Database cluster. The command also updates the `PYTHONPATH`, `PATH`, and `LD_LIBRARY_PATH` environment variables in your `greenplum_path.sh` file.

4. Restart Greenplum Database. You must re-source `greenplum_path.sh` before restarting your Greenplum cluster:

```
$ source /usr/local/greenplum-db/greenplum_path.sh
$ gpstop -r
```

The Greenplum Database Python Data Science Modules are installed in the following directory:

```
$GPHOME/ext/DataSciencePython/lib/python2.7/site-packages/
```

Uninstalling the Python Data Science Module Package

Use the `gppkg` utility to uninstall the Python Data Science Module package. You must include the version number in the package name you provide to `gppkg`.

To determine your Python Data Science Module package version number and remove this package:

```
$ gppkg -q --all | grep DataSciencePython
DataSciencePython-<version>
$ gppkg -r DataSciencePython-<version>
```

The command removes the Python Data Science modules from your Greenplum Database cluster. It also updates the `PYTHONPATH`, `PATH`, and `LD_LIBRARY_PATH` environment variables in your `greenplum_path.sh` file to their pre-installation values.

Re-source `greenplum_path.sh` and restart Greenplum Database after you remove the Python Data Science Module package:

```
$ . /usr/local/greenplum-db/greenplum_path.sh
$ gpstop -r
```

> **Note:** When you uninstall the Python Data Science Module package from your Greenplum Database cluster, any UDFs that you have created that import Python modules installed with this package will return an error.

R Data Science Library Package

R packages are modules that contain R functions and data sets. Greenplum Database provides a collection of data science-related R libraries that can be used with the Greenplum Database PL/R language. You can download these libraries in `.gppkg` format from *Pivotal Network*.

This chapter contains the following information:

- *R Data Science Libraries*
- *Installing the R Data Science Library Package*
- *Uninstalling the R Data Science Library Package*

For information about the Greenplum Database PL/R Language, see *Greenplum PL/R Language Extension*.

R Data Science Libraries

Libraries provided in the R Data Science package include:

abind	gss	R2WinBUGS
adabag	gtable	R6
arm	gtools	randomForest

assertthat	hms	RColorBrewer
backports	hybridHclust	Rcpp
BH	igraph	RcppArmadillo
bitops	ipred	RcppEigen
car	iterators	RcppRoll
caret	labeling	readr
caTools	lattice	recipes
cli	lava	reshape2
clipr	lazyeval	rjags
coda	lme4	rlang
colorspace	lmtest	RobustRankAggreg
compHclust	lubridate	ROCR
crayon	magrittr	rpart
curl	MASS	RPostgreSQL
data.table	Matrix	sandwich
DBI	MatrixModels	scales
Deriv	mcmc	SparseM
dichromat	MCMCpack	SQUAREM
digest	minqa	stabledist
doParallel	ModelMetrics	stringi
dplyr	MTS	stringr
e1071	munsell	survival
fansi	mvtnorm	tibble
fastICA	neuralnet	tidyr
fBasics	nloptr	tidyselect
fGarch	nnet	timeDate
flashClust	numDeriv	timeSeries
foreach	pbkrtest	tseries
forecast	pillar	TTR
foreign	pkgconfig	urca
fracdiff	plogr	utf8
gdata	plyr	vctrs
generics	prodlim	viridisLite
ggplot2	purrr	withr
glmnet	quadprog	xts
glue	quantmod	zeallot
gower	quantreg	zoo

gplots	R2jags	

Installing the R Data Science Library Package

Before you install the R Data Science Library package, make sure that your Greenplum Database is running, you have sourced `greenplum_path.sh`, and that the `$MASTER_DATA_DIRECTORY` and `$GPHOME` environment variables are set.

1. Locate the R Data Science library package that you built or downloaded.

   The file name format of the package is `DataScienceR-<version>-relhel<N>_x86_64.gppkg`.
2. Copy the package to the Greenplum Database master host.
3. Use the `gppkg` command to install the package. For example:

   ```
   $ gppkg -i DataScienceR-<version>-relhel<N>_x86_64.gppkg
   ```

   `gppkg` installs the R Data Science libraries on all nodes in your Greenplum Database cluster. The command also sets the `R_LIBS_USER` environment variable and updates the `PATH` and `LD_LIBRARY_PATH` environment variables in your `greenplum_path.sh` file.
4. Restart Greenplum Database. You must re-source `greenplum_path.sh` before restarting your Greenplum cluster:

   ```
   $ source /usr/local/greenplum-db/greenplum_path.sh
   $ gpstop -r
   ```

The Greenplum Database R Data Science Modules are installed in the following directory:

```
$GPHOME/ext/DataScienceR/library
```

> **Note:** `rjags` libraries are installed in the `$GPHOME/ext/DataScienceR/extlib/lib` directory. If you want to use `rjags` and your `$GPHOME` is not `/usr/local/greenplum-db`, you must perform additional configuration steps to create a symbolic link from `$GPHOME` to `/usr/local/greenplum-db` on each node in your Greenplum Database cluster. For example:
>
> ```
> $ gpssh -f all_hosts -e 'ln -s $GPHOME /usr/local/greenplum-db'
> $ gpssh -f all_hosts -e 'chown -h gpadmin /usr/local/greenplum-db'
> ```

Uninstalling the R Data Science Library Package

Use the `gppkg` utility to uninstall the R Data Science Library package. You must include the version number in the package name you provide to `gppkg`.

To determine your R Data Science Library package version number and remove this package:

```
$ gppkg -q --all | grep DataScienceR
DataScienceR-<version>
$ gppkg -r DataScienceR-<version>
```

The command removes the R Data Science libraries from your Greenplum Database cluster. It also removes the `R_LIBS_USER` environment variable and updates the `PATH` and `LD_LIBRARY_PATH` environment variables in your `greenplum_path.sh` file to their pre-installation values.

Re-source `greenplum_path.sh` and restart Greenplum Database after you remove the R Data Science Library package:

```
$ . /usr/local/greenplum-db/greenplum_path.sh
$ gpstop -r
```

Note: When you uninstall the R Data Science Library package from your Greenplum Database cluster, any UDFs that you have created that use R libraries installed with this package will return an error.

Greenplum Platform Extension Framework (PXF)

Optional. If you do not plan to use PXF, no action is necessary.

If you plan to use PXF, refer to *Accessing External Data with PXF* for introductory PXF information.

Installing Additional Supplied Modules

The Greenplum Database distribution includes several PostgreSQL- and Greenplum-sourced `contrib` modules that you have the option to install.

Each module is typically packaged as a Greenplum Database extension. You must register these modules in each database in which you want to use it. For example, to register the `dblink` module in the database named `testdb`, use the command:

```
$ psql -d testdb -c 'CREATE EXTENSION dblink;'
```

To remove a module from a database, drop the associated extension. For example, to remove the `dblink` module from the `testdb` database:

```
$ psql -d testdb -c 'DROP EXTENSION dblink;'
```

> **Note:** When you drop a module extension from a database, any user-defined function that you created in the database that references functions defined in the module will no longer work. If you created any database objects that use data types defined in the module, Greenplum Database will notify you of these dependencies when you attempt to drop the module extension.

You can register the following modules in this manner:

For additional information about the modules supplied with Greenplum Database, refer to *Additional Supplied Modules* in the *Greenplum Database Reference Guide*.

Configuring Timezone and Localization Settings

Describes the available timezone and localization features of Greenplum Database.

Configuring the Timezone

Greenplum Database selects a timezone to use from a set of internally stored PostgreSQL timezones. The available PostgreSQL timezones are taken from the Internet Assigned Numbers Authority (IANA) Time Zone Database, and Greenplum Database updates its list of available timezones as necessary when the IANA database changes for PostgreSQL.

Greenplum Database selects the timezone by matching a PostgreSQL timezone with the value of the `TimeZone` server configuration parameter, or the host system time zone if `TimeZone` is not set. For example, when selecting a default timezone from the host system time zone, Greenplum Database uses an algorithm to select a PostgreSQL timezone based on the host system timezone files. If the system timezone includes leap second information, Greenplum Database cannot match the system timezone with a PostgreSQL timezone. In this case, Greenplum Database calculates a "best match" with a PostgreSQL timezone based on information from the host system.

As a best practice, configure Greenplum Database and the host systems to use a known, supported timezone. This sets the timezone for the Greenplum Database master and segment instances, and prevents Greenplum Database from selecting a best match timezone each time the cluster is restarted, using the current system timezone and Greenplum Database timezone files (which may have been updated from the IANA database since the last restart). Use the `gpconfig` utility to show and set the Greenplum Database timezone. For example, these commands show the Greenplum Database timezone and set the timezone to `US/Pacific`.

```
# gpconfig -s TimeZone
# gpconfig -c TimeZone -v 'US/Pacific'
```

You must restart Greenplum Database after changing the timezone. The command `gpstop -ra` restarts Greenplum Database. The catalog view `pg_timezone_names` provides Greenplum Database timezone information.

About Locale Support in Greenplum Database

Greenplum Database supports localization with two approaches:

- Using the locale features of the operating system to provide locale-specific collation order, number formatting, and so on.
- Providing a number of different character sets defined in the Greenplum Database server, including multiple-byte character sets, to support storing text in all kinds of languages, and providing character set translation between client and server.

Locale support refers to an application respecting cultural preferences regarding alphabets, sorting, number formatting, etc. Greenplum Database uses the standard ISO C and POSIX locale facilities provided by the server operating system. For additional information refer to the documentation of your operating system.

Locale support is automatically initialized when a Greenplum Database system is initialized. The initialization utility, *`gpinitsystem`*, will initialize the Greenplum array with the locale setting of its execution environment by default, so if your system is already set to use the locale that you want in your Greenplum Database system then there is nothing else you need to do.

When you are ready to initiate Greenplum Database and you want to use a different locale (or you are not sure which locale your system is set to), you can instruct `gpinitsystem` exactly which locale to use by specifying the -n *locale* option. For example:

```
$ gpinitsystem -c gp_init_config -n sv_SE
```

See *Initializing a Greenplum Database System* for information about the database initialization process.

The example above sets the locale to Swedish (sv) as spoken in Sweden (SE). Other possibilities might be `en_US` (U.S. English) and `fr_CA` (French Canadian). If more than one character set can be useful for a locale then the specifications look like this: `cs_CZ.ISO8859-2`. What locales are available under what names on your system depends on what was provided by the operating system vendor and what was installed. On most systems, the command `locale -a` will provide a list of available locales.

Occasionally it is useful to mix rules from several locales, for example use English collation rules but Spanish messages. To support that, a set of locale subcategories exist that control only a certain aspect of the localization rules:

- `LC_COLLATE` — String sort order
- `LC_CTYPE` — Character classification (What is a letter? Its upper-case equivalent?)
- `LC_MESSAGES` — Language of messages
- `LC_MONETARY` — Formatting of currency amounts
- `LC_NUMERIC` — Formatting of numbers
- `LC_TIME` — Formatting of dates and times

If you want the system to behave as if it had no locale support, use the special locale `C` or `POSIX`.

The nature of some locale categories is that their value has to be fixed for the lifetime of a Greenplum Database system. That is, once `gpinitsystem` has run, you cannot change them anymore. `LC_COLLATE` and `LC_CTYPE` are those categories. They affect the sort order of indexes, so they must be kept fixed, or indexes on text columns will become corrupt. Greenplum Database enforces this by recording the values of `LC_COLLATE` and `LC_CTYPE` that are seen by gpinitsystem. The server automatically adopts those two values based on the locale that was chosen at initialization time.

The other locale categories can be changed as desired whenever the server is running by setting the server configuration parameters that have the same name as the locale categories (see the *Greenplum Database Reference Guide* for more information on setting server configuration parameters). The defaults that are chosen by gpinitsystem are written into the master and segment `postgresql.conf` configuration files to serve as defaults when the Greenplum Database system is started. If you delete these assignments from the master and each segment `postgresql.conf` files then the server will inherit the settings from its execution environment.

Note that the locale behavior of the server is determined by the environment variables seen by the server, not by the environment of any client. Therefore, be careful to configure the correct locale settings on each Greenplum Database host (master and segments) before starting the system. A consequence of this is that if client and server are set up in different locales, messages may appear in different languages depending on where they originated.

Inheriting the locale from the execution environment means the following on most operating systems: For a given locale category, say the collation, the following environment variables are consulted in this order until one is found to be set: `LC_ALL`, `LC_COLLATE` (the variable corresponding to the respective category), `LANG`. If none of these environment variables are set then the locale defaults to `C`.

Some message localization libraries also look at the environment variable `LANGUAGE` which overrides all other locale settings for the purpose of setting the language of messages. If in doubt, please refer to the documentation for your operating system, in particular the documentation about `gettext`, for more information.

Native language support (NLS), which enables messages to be translated to the user's preferred language, is not enabled in Greenplum Database for languages other than English. This is independent of the other locale support.

Locale Behavior

The locale settings influence the following SQL features:

- Sort order in queries using `ORDER BY` on textual data
- The ability to use indexes with `LIKE` clauses
- The `upper`, `lower`, and `initcap` functions
- The `to_char` family of functions

The drawback of using locales other than `C` or `POSIX` in Greenplum Database is its performance impact. It slows character handling and prevents ordinary indexes from being used by `LIKE`. For this reason use locales only if you actually need them.

Troubleshooting Locales

If locale support does not work as expected, check that the locale support in your operating system is correctly configured. To check what locales are installed on your system, you may use the command `locale -a` if your operating system provides it.

Check that Greenplum Database is actually using the locale that you think it is. `LC_COLLATE` and `LC_CTYPE` settings are determined at initialization time and cannot be changed without redoing *`gpinitsystem`*. Other locale settings including `LC_MESSAGES` and `LC_MONETARY` are initially determined by the operating system environment of the master and/or segment host, but can be changed after initialization by editing the `postgresql.conf` file of each Greenplum master and segment instance. You can check the active locale settings of the master host using the `SHOW` command. Note that every host in your Greenplum Database array should be using identical locale settings.

Character Set Support

The character set support in Greenplum Database allows you to store text in a variety of character sets, including single-byte character sets such as the ISO 8859 series and multiple-byte character sets such as EUC (Extended Unix Code), UTF-8, and Mule internal code. All supported character sets can be used transparently by clients, but a few are not supported for use within the server (that is, as a server-side encoding). The default character set is selected while initializing your Greenplum Database array using `gpinitsystem`. It can be overridden when you create a database, so you can have multiple databases each with a different character set.

Table 9: Greenplum Database Character Sets

Name	Description	Language	Server?	Bytes/Char	Aliases
BIG5	Big Five	Traditional Chinese	No	1-2	WIN950, Windows950
EUC_CN	Extended UNIX Code-CN	Simplified Chinese	Yes	1-3	
EUC_JP	Extended UNIX Code-JP	Japanese	Yes	1-3	
EUC_KR	Extended UNIX Code-KR	Korean	Yes	1-3	

[1] **Not all APIs support all the listed character sets. For example, the JDBC driver does not support MULE_INTERNAL, LATIN6, LATIN8, and LATIN10.**

Name	Description	Language	Server?	Bytes/Char	Aliases
EUC_TW	Extended UNIX Code-TW	Traditional Chinese, Taiwanese	Yes	1-3	
GB18030	National Standard	Chinese	No	1-2	
GBK	Extended National Standard	Simplified Chinese	No	1-2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	Latin/Cyrillic	Yes	1	
ISO_8859_6	ISO 8859-6, ECMA 114	Latin/Arabic	Yes	1	
ISO_8859_7	ISO 8859-7, ECMA 118	Latin/Greek	Yes	1	
ISO_8859_8	ISO 8859-8, ECMA 121	Latin/Hebrew	Yes	1	
JOHAB	JOHA	Korean (Hangul)	Yes	1-3	
KOI8	KOI8-R(U)	Cyrillic	Yes	1	KOI8R
LATIN1	ISO 8859-1, ECMA 94	Western European	Yes	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	Central European	Yes	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	South European	Yes	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	North European	Yes	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	Turkish	Yes	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Nordic	Yes	1	ISO885910
LATIN7	ISO 8859-13	Baltic	Yes	1	ISO885913
LATIN8	ISO 8859-14	Celtic	Yes	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 with Euro and accents	Yes	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Romanian	Yes	1	ISO885916
MULE_ INTERNAL	Mule internal code	Multilingual Emacs	Yes	1-4	

Name	Description	Language	Server?	Bytes/Char	Aliases
SJIS	Shift JIS	Japanese	No	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SQL_ASCII	unspecified[2]	any	No	1	
UHC	Unified Hangul Code	Korean	No	1-2	WIN949, Windows949
UTF8	Unicode, 8-bit	all	Yes	1-4	Unicode
WIN866	Windows CP866	Cyrillic	Yes	1	ALT
WIN874	Windows CP874	Thai	Yes	1	
WIN1250	Windows CP1250	Central European	Yes	1	
WIN1251	Windows CP1251	Cyrillic	Yes	1	WIN
WIN1252	Windows CP1252	Western European	Yes	1	
WIN1253	Windows CP1253	Greek	Yes	1	
WIN1254	Windows CP1254	Turkish	Yes	1	
WIN1255	Windows CP1255	Hebrew	Yes	1	
WIN1256	Windows CP1256	Arabic	Yes	1	
WIN1257	Windows CP1257	Baltic	Yes	1	
WIN1258	Windows CP1258	Vietnamese	Yes	1	ABC, TCVN, TCVN5712, VSCII

Setting the Character Set

gpinitsystem defines the default character set for a Greenplum Database system by reading the setting of the `ENCODING` parameter in the `gp_init_config` file at initialization time. The default character set is `UNICODE` or `UTF8`.

[2] The SQL_ASCII setting behaves considerably differently from the other settings. Byte values 0-127 are interpreted according to the ASCII standard, while byte values 128-255 are taken as uninterpreted characters. If you are working with any non-ASCII data, it is unwise to use the SQL_ASCII setting as a client encoding. SQL_ASCII is not supported as a server encoding.

You can create a database with a different character set besides what is used as the system-wide default. For example:

```
=> CREATE DATABASE korean WITH ENCODING 'EUC_KR';
```

> **Important:** Although you can specify any encoding you want for a database, it is unwise to choose an encoding that is not what is expected by the locale you have selected. The `LC_COLLATE` and `LC_CTYPE` settings imply a particular encoding, and locale-dependent operations (such as sorting) are likely to misinterpret data that is in an incompatible encoding.

Since these locale settings are frozen by gpinitsystem, the apparent flexibility to use different encodings in different databases is more theoretical than real.

One way to use multiple encodings safely is to set the locale to `C` or `POSIX` during initialization time, thus disabling any real locale awareness.

Character Set Conversion Between Server and Client

Greenplum Database supports automatic character set conversion between server and client for certain character set combinations. The conversion information is stored in the master `pg_conversion` system catalog table. Greenplum Database comes with some predefined conversions or you can create a new conversion using the SQL command `CREATE CONVERSION`.

Table 10: Client/Server Character Set Conversions

Server Character Set	Available Client Character Sets
BIG5	not supported as a server encoding
EUC_CN	EUC_CN, MULE_INTERNAL, UTF8
EUC_JP	EUC_JP, MULE_INTERNAL, SJIS, UTF8
EUC_KR	EUC_KR, MULE_INTERNAL, UTF8
EUC_TW	EUC_TW, BIG5, MULE_INTERNAL, UTF8
GB18030	not supported as a server encoding
GBK	not supported as a server encoding
ISO_8859_5	ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	ISO_8859_6, UTF8
ISO_8859_7	ISO_8859_7, UTF8
ISO_8859_8	ISO_8859_8, UTF8
JOHAB	JOHAB, UTF8
KOI8	KOI8, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
LATIN1	LATIN1, MULE_INTERNAL, UTF8
LATIN2	LATIN2, MULE_INTERNAL, UTF8, WIN1250
LATIN3	LATIN3, MULE_INTERNAL, UTF8
LATIN4	LATIN4, MULE_INTERNAL, UTF8
LATIN5	LATIN5, UTF8

Server Character Set	Available Client Character Sets
LATIN6	LATIN6, UTF8
LATIN7	LATIN7, UTF8
LATIN8	LATIN8, UTF8
LATIN9	LATIN9, UTF8
LATIN10	LATIN10, UTF8
MULE_INTERNAL	MULE_INTERNAL, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	not supported as a server encoding
SQL_ASCII	not supported as a server encoding
UHC	not supported as a server encoding
UTF8	all supported encodings
WIN866	WIN866
ISO_8859_5	KOI8, MULE_INTERNAL, UTF8, WIN1251
WIN874	WIN874, UTF8
WIN1250	WIN1250, LATIN2, MULE_INTERNAL, UTF8
WIN1251	WIN1251, ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866
WIN1252	WIN1252, UTF8
WIN1253	WIN1253, UTF8
WIN1254	WIN1254, UTF8
WIN1255	WIN1255, UTF8
WIN1256	WIN1256, UTF8
WIN1257	WIN1257, UTF8
WIN1258	WIN1258, UTF8

To enable automatic character set conversion, you have to tell Greenplum Database the character set (encoding) you would like to use in the client. There are several ways to accomplish this:

- Using the `\encoding` command in `psql`, which allows you to change client encoding on the fly.
- Using `SET client_encoding TO`. Setting the client encoding can be done with this SQL command:

  ```
  => SET CLIENT_ENCODING TO 'value';
  ```

  To query the current client encoding:

  ```
  => SHOW client_encoding;
  ```

  To return to the default encoding:

  ```
  => RESET client_encoding;
  ```

- Using the `PGCLIENTENCODING` environment variable. When `PGCLIENTENCODING` is defined in the client's environment, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)
- Setting the configuration parameter `client_encoding`. If `client_encoding` is set in the master `postgresql.conf` file, that client encoding is automatically selected when a connection to Greenplum Database is made. (This can subsequently be overridden using any of the other methods mentioned above.)

If the conversion of a particular character is not possible — suppose you chose `EUC_JP` for the server and `LATIN1` for the client, then some Japanese characters do not have a representation in `LATIN1` — then an error is reported.

If the client character set is defined as `SQL_ASCII`, encoding conversion is disabled, regardless of the server's character set. The use of `SQL_ASCII` is unwise unless you are working with all-ASCII data. `SQL_ASCII` is not supported as a server encoding.

Upgrading to Greenplum 6

This topic identifies the upgrade and migration paths supported for the Greenplum Database 6.x. release.

Greenplum Database 6 supports upgrading from a Greenplum 6.x release to a newer Greenplum 6.x release. Direct upgrade from Greenplum Database 4.3 or 5 to Greenplum 6 is not supported; you must migrate the data to Greenplum 6.

Upgrading from an Earlier Greenplum 6 Release

The upgrade path supported for this release is Greenplum Database 6.x to a newer Greenplum Database 6.x release.

> **Important:** Set the Greenplum Database timezone to a value that is compatible with your host systems. Setting the Greenplum Database timezone prevents Greenplum Database from selecting a timezone each time the cluster is restarted and sets the timezone for the Greenplum Database master and segment instances. After you upgrade to this release and if you have not set a Greenplum Database timezone value, verify that the selected Greenplum Database timezone is acceptable for your deployment. See *Configuring Timezone and Localization Settings* for more information.

Prerequisites

Before starting the upgrade process, perform the following checks.

- Verify the health of the Greenplum Database host hardware, and verify that the hosts meet the requirements for running Greenplum Database. The Greenplum Database `gpcheckperf` utility can assist you in confirming the host requirements.

  > **Note:** If you need to run the `gpcheckcat` utility, run it a few weeks before the upgrade during a maintenance period. If necessary, you can resolve any issues found by the utility before the scheduled upgrade.

  The utility is in `$GPHOME/bin`. Place Greenplum Database in restricted mode when you run the `gpcheckcat` utility. See the *Greenplum Database Utility Guide* for information about the `gpcheckcat` utility.

  If `gpcheckcat` reports catalog inconsistencies, you can run `gpcheckcat` with the `-g` option to generate SQL scripts to fix the inconsistencies.

  After you run the SQL scripts, run `gpcheckcat` again. You might need to repeat the process of running `gpcheckcat` and creating SQL scripts to ensure that there are no inconsistencies. Run the SQL scripts generated by `gpcheckcat` on a quiescent system. The utility might report false alerts if there is activity on the system.

  > **Important:** If the `gpcheckcat` utility reports errors, but does not generate a SQL script to fix the errors, contact Pivotal Support. Information for contacting Pivotal Support is at *https://support.pivotal.io*.

- If you have configured the Greenplum Platform Extension Framework (PXF) in your previous Greenplum Database installation, you must stop the PXF service, and you might need to back up PXF configuration files before upgrading to a new version of Greenplum Database. Refer to *PXF Pre-Upgrade Actions* for instructions.

  If you have not yet configured PXF, no action is necessary.

- If you have configured and used the Greenplum Streaming Server (GPSS) in your previous Greenplum Database installation, you must stop any running GPSS jobs and service instances before you upgrade to a new version of Greenplum Database. Refer to *GPSS Pre-Upgrade Actions* for instructions.

  If you do not plan to use GPSS, or you have not yet configured GPSS, no action is necessary.

Upgrading from 6.x to a Newer 6.x Release

An upgrade from Greenplum Database 6.x to a newer 6.x release involves stopping Greenplum Database, updating the Greenplum Database software binaries, and restarting Greenplum Database. If you are using Greenplum Database extension packages there are additional requirements. See *Prerequisites* in the previous section.

1. Log in to your Greenplum Database master host as the Greenplum administrative user:

   ```
   $ su - gpadmin
   ```

2. Perform a smart shutdown of your Greenplum Database 6.x system (there can be no active connections to the database). This example uses the `-a` option to disable confirmation prompts:

   ```
   $ gpstop -a
   ```

3. Copy the new Greenplum Database software installation package to the `gpadmin` user's home directory on each master, standby, and segment host.
4. *If you used `yum` or `apt` to install Greenplum Database to the default location*, execute these commands on each host to upgrade to the new software release.

   For RHEL/CentOS systems:

   ```
   $ sudo yum upgrade ./greenplum-db-<version>-<platform>.rpm
   ```

   For Ubuntu systems:

   ```
   # apt install ./greenplum-db-<version>-<platform>.deb
   ```

   The `yum` or `apt` command installs the new Greenplum Database software files into a version-specific directory under `/usr/local` and updates the symbolic link `/usr/local/greenplum-db` to point to the new installation directory.
5. *If you used `rpm` to install Greenplum Database to a non-default location on RHEL/CentOS systems*, execute `rpm` on each host to upgrade to the new software release and specify the same custom installation directory with the `--prefix` option. For example:

   ```
   $ sudo rpm -U ./greenplum-db-<version>-<platform>.rpm --prefix=<directory>
   ```

   The `rpm` command installs the new Greenplum Database software files into a version-specific directory under the `<directory>` you specify, and updates the symbolic link `<directory>/greenplum-db` to point to the new installation directory.
6. Update the permissions for the new installation. For example, run this command as `root` to change the user and group of the installed files to `gpadmin`.

   ```
   $ sudo chown -R gpadmin:gpadmin /usr/local/greenplum*
   ```

7. If needed, update the `greenplum_path.sh` file on the master and standby master hosts for use with your specific installation. These are some examples.

   - If Greenplum Database uses LDAP authentication, edit the `greenplum_path.sh` file to add the line:

     ```
     export LDAPCONF=/etc/openldap/ldap.conf
     ```

   - If Greenplum Database uses PL/Java, you might need to set or update the environment variables `JAVA_HOME` and `LD_LIBRARY_PATH` in `greenplum_path.sh`.

     Note: When comparing the previous and new `greenplum_path.sh` files, be aware that installing some Greenplum Database extensions also updates the `greenplum_path.sh` file.

The `greenplum_path.sh` from the previous release might contain updates that were the result of installing those extensions.

8. Edit the environment of the Greenplum Database superuser (`gpadmin`) and make sure you are sourcing the `greenplum_path.sh` file for the new installation. For example change the following line in the `.bashrc` or your chosen profile file:

   ```
   source /usr/local/greenplum-db-<current_version>/greenplum_path.sh
   ```

   to:

   ```
   source /usr/local/greenplum-db-<new_version>/greenplum_path.sh
   ```

   Or if you are sourcing a symbolic link (`/usr/local/greenplum-db`) in your profile files, update the link to point to the newly installed version. For example:

   ```
   $ rm /usr/local/greenplum-db
   $ ln -s /usr/local/greenplum-db-<new_version> /usr/local/greenplum-db
   ```

9. Source the environment file you just edited. For example:

   ```
   $ source ~/.bashrc
   ```

10. Use the Greenplum Database `gppkg` utility to re-install Greenplum Database extensions. If you were previously using any Greenplum Database extensions such as pgcrypto, PL/R, PL/Java, or PostGIS, download the corresponding packages from *Pivotal Network*, and install using this utility. See the extension documentation for details.

    Also copy any files that are used by the extensions (such as JAR files, shared object files, and libraries) from the previous version installation directory to the new version installation directory on the master and segment host systems.

11. After all segment hosts have been upgraded, log in as the `gpadmin` user and restart your Greenplum Database system:

    ```
    # su - gpadmin
    $ gpstart
    ```

12. If you configured PXF in your previous Greenplum Database installation, you may need to install PXF in your new Greenplum installation, and you may be required to re-initialize the PXF service after you upgrade Greenplum Database. Refer to the *Step 2* PXF upgrade procedure for instructions.
13. If you configured GPSS in your previous Greenplum Database installation, you may be required to perform some upgrade actions, and you must re-restart the GPSS service instances and jobs. Refer to *Step 2* of the GPSS upgrade procedure for instructions.

After upgrading Greenplum Database, ensure that all features work as expected. For example, test that backup and restore perform as expected, and Greenplum Database features such as user-defined functions, and extensions such as MADlib and PostGIS perform as expected.

Troubleshooting a Failed Upgrade

If you experience issues during the migration process and have active entitlements for Greenplum Database that were purchased through Pivotal, contact Pivotal Support. Information for contacting Pivotal Support is at *https://support.pivotal.io*.

Be prepared to provide the following information:

- A completed *Upgrade Procedure*
- Log output from `gpcheckcat` (located in `~/gpAdminLogs`)

Migrating Data from Greenplum 4.3 or 5 to Greenplum 6

You can migrate data from Greenplum Database 4.3 or 5 to Greenplum 6 using the standard backup and restore procedures, `gpbackup` and `gprestore`, or by using `gpcopy`.

> **Note:** Currently, you cannot upgrade a Greenplum Database 4.3 or 5 system directly to Greenplum Database 6.

This topic identifies known issues you may encounter when moving data from Greenplum 4.3 to Greenplum 6. You can work around these problems by making needed changes to your Greenplum 4.3 databases so that you can create backups that can be restored successfully to Greenplum 6.

Preparing the Greenplum 6 Cluster

- Install and initialize a new Greenplum Database 6 cluster using the version 6 `gpinitsystem` utility.

  > **Note:** `gprestore` only supports restoring data to a cluster that has an identical number of hosts and an identical number of segments per host, with each segment having the same `content_id` as the segment in the original cluster. Use `gpcopy` if you need to migrate data to a different-sized Greenplum 6 cluster.

  > **Note:** Set the Greenplum Database 6 timezone to a value that is compatible with your host systems. Setting the Greenplum Database timezone prevents Greenplum Database from selecting a timezone each time the cluster is restarted. See *Configuring Timezone and Localization Settings* for more information.
- Install the latest release of the Greenplum Backup and Restore utilities, available to download from *Pivotal Network*.
- If you intend to install Greenplum Database 6 on the same hardware as your 4.3 system, you will need enough disk space to accommodate over five times the original data set (two full copies of the primary and mirror data sets, plus the original backup data in ASCII format) in order to migrate data with `gpbackup` and `gprestore`. Keep in mind that the ASCII backup data will require more disk space than the original data, which may be stored in compressed binary format. Offline backup solutions such as Dell EMC Data Domain can reduce the required disk space on each host.

  If you want to migrate your data on the same hardware but do not have enough free disk space on your host systems, `gpcopy` provides the `--truncate-source-after` option to truncate each source table after copying the table to the destination cluster and validating that the copy succeeded. This reduces the amount of free space needed to migrate clusters that reside on the same hardware. See *Migrating Data with gpcopy* for more information.
- Install any external modules used in your Greenplum 4.3 system in the Greenplum 6 system before you restore the backup, for example MADlib or PostGIS. If versions of the external modules are not compatible, you may need to exclude tables that reference them when restoring the Greenplum 4.3 backup to Greenplum 6.
- The Greenplum 4.3 Oracle Compatibility Functions module is not compatible with Greenplum 6. Uninstall the module from Greenplum 4.3 by running the `uninstall_orafunc.sql` script before you back up your databases:

  ```
  $ $GPHOME/share/postgresql/contrib/uninstall_orafunc.sql
  ```

  You will also need to drop any dependent database objects that reference compatibility functions.

Install the Oracle Compatibility Functions in Greenplum 6 by creating the `orafce` module in each database where you require the functions:

```
$ psql -d dbname 'CREATE EXTENSION orafce'
```

See *Installing Additional Supplied Modules* for information about installing `orafce` and other modules.

- When restoring language-based user-defined functions, the shared object file must be in the location specified in the `CREATE FUNCTION` SQL command and must have been recompiled on the Greenplum 6 system. This applies to user-defined functions, user-defined types, and any other objects that use custom functions, such as aggregates created with the `CREATE AGGREGATE` command.
- Greenplum 6 provides *resource groups*, an alternative to managing resources using resource queues. Setting the `gp_resource_manager` server configuration parameter to `queue` or `group` selects the resource management scheme the Greenplum Database system will use. The default is `queue`, so no action is required when you move from Greenplum version 4.3 to version 6. To more easily transition from resource queues to resource groups, you can set resource groups to allocate and manage memory in a way that is similar to resource queue memory management. To select this feature, set the `MEMORY_LIMIT` and `MEMORY_SPILL_RATIO` attributes of your resource groups to 0. See *Using Resource Groups* for information about enabling and configuring resource groups.

Preparing Greenplum 4.3 and 5 Databases for Backup

Note: A Greenplum 4 system must be at least version 4.3.22 to use the `gpbackup` and `gprestore` utilities. A Greenplum 5 system must be at least version 5.5. Be sure to use the latest release of the backup and restore utilities, available for download from *Pivotal Network*.

Important: Make sure that you have a complete backup of all data in the Greenplum Database 4.3 or 5 cluster, and that you can successfully restore the Greenplum Database cluster if necessary.

Following are some issues that are known to cause errors when restoring a Greenplum 4.3 or 5 backup to Greenplum 6. Keep a list of any changes you make to the Greenplum 4.3 or 5 database to enable migration so that you can fix them in Greenplum 6 after restoring the backup.

- If you have configured PXF in your Greenplum Database 5 installation, review *Migrating PXF from Greenplum 5* to plan for the PXF migration.
- Greenplum Database version 6 removes support for the `gphdfs` protocol. If you have created external tables that use gphdfs, remove the external table definitions and (optionally) recreate them to use Pivotal Extension Framework (PXF) before you migrate the data to Greenplum 6. Refer to *Migrating gphdfs External Tables to PXF* in the PXF documentation for the migration procedure.
- References to catalog tables or their attributes can cause a restore to fail due to catalog changes from Greenplum 4.3 or 5 to Greenplum 6. Here are some catalog issues to be aware of when migrating to Greenplum 6:
  - In the `pg_class` system table, the `reltoastidxid` column has been removed.
  - In the `pg_stat_replication` system table, the `procpid` column is renamed to `pid`.
  - In the `pg_stat_activity` system table, the `procpid` column is renamed to `pid`. The `current_query` column is replaced by two columns `state`: the state of the backend, and `query`: the last run query, or currently running query if `state` is `active`.
  - In the `gp_distribution_policy` system table, the `attrnums` column is renamed to `distkey` and its data type is changed to `int2vector`. A backend function `pg_get_table_distributedby()` was added to get the distribution policy for a table as a string.
  - The `__gp_localid` and `__gp_masterid` columns are removed from the `session_level_memory_consumption` system view in Greenplum 6. The underlying external tables and functions are removed from the `gp_toolkit` schema.
  - Filespaces are removed in Greenplum 6. The `pg_filespace` and `pg_filespace_entry` system tables are removed. Any reference to `pg_filespace` or `pg_filespace_entry` will fail in Greenplum 6.

    - Restoring a Greenplum 4 backup can fail due to lack of dependency checking in Greenplum 4 catalog tables. For example, restoring a UDF can fail if it references a custom data type that is created later in the backup file.
- The `INTO error_table` clause of the `CREATE EXTERNAL TABLE` and `COPY` commands was deprecated in Greenplum 4.3 and is unsupported in Greenplum 5 and 6. Remove this clause from any external table definitions before you create a backup of your Greenplum 4.3 system. The `ERROR_TABLE` parameter of the `gpload` utility load control YAML file must also be removed from any `gpload` YAML files before you run `gpload`.
- The `int4_avg_accum()` function signature changed in Greenplum 6 from `int4_avg_accum(bytea, integer)` to `int4_avg_accum(bigint[], integer)`. This function is the state transition function (*sfunc*) called when calculating the average of a series of 4-byte integers. If you have created a custom aggregate in a previous Greenplum release that called the built-in `int4_avg_accum()` function, you will need to revise your aggregate for the new signature.
- The `string_agg(expression)` function has been removed from Greenplum 6. The function concatenates text values into a string. You can replace the single argument function with the function `string_agg(expression, delimiter)` and specify an empty string as the `delimiter`, for example `string_agg(txt_col1, '')`.
- The `offset` argument of the `lag(expr, offset[, default])` window function has changed from `int8` in Greenplum 4.3 to `int4` in Greenplum 5 and 6.
- `gpbackup` saves the distribution policy and distribution key for each table in the backup so that data can be restored to the same segment. If a table's distribution key in the Greenplum 4.3 or 5 database is incompatible with Greenplum 6, `gprestore` cannot restore the table to the correct segment in the Greenplum 6 database. This can happen if the distribution key in the older Greenplum release has columns with data types not allowed in Greenplum 6 distribution keys, or if the data representation for data types has changed or is insufficient for Greenplum 6 to generate the same hash value for a distribution key. You should correct these kinds of problems by altering distribution keys in the tables before you back up the Greenplum database.
- Greenplum 6 requires primary keys and unique index keys to match a table's distribution key. The leaf partitions of partitioned tables must have the same distribution policy as the root partition. These known issues should be corrected in the source Greenplum database before you back up the database:
    - If the primary key is different than the distribution key for a table, alter the table to either remove the primary key or change the primary key to match the distribution key.
    - If the key columns for a unique index are not a subset of the distribution key columns, before you back up the source database, drop the index and, optionally, recreate it with a compatible key.
    - If a partitioned table in the source database has a `DISTRIBUTED BY` distribution policy, but has leaf partitions that are `DISTRIBUTED RANDOMLY`, alter the leaf tables to match the root table distribution policy before you back up the source database.
- In Greenplum 4.3, the name provided for a constraint in a `CREATE TABLE` command was the name of the index created to enforce the constraint, which could lead to indexes having the same name. In Greenplum 6, duplicate index names are not allowed; restoring from a Greenplum 4.3 backup that has duplicate index names will generate errors.
- Columns of type `abstime`, `reltime`, `tinterval`, `money`, or `anyarray` are not supported as distribution keys in Greenplum 6.

  If you have tables distributed on columns of type `abstime`, `reltime`, `tinterval`, `money`, or `anyarray`, use the `ALTER TABLE` command to set the distribution to `RANDOM` before you back up the database. After the data is restored, you can set a new distribution policy.
- In Greenplum 4.3 and 5, it was possible to `ALTER` a table that has a primary key or unique index to be `DISTRIBUTED RANDOMLY`. Greenplum 6 does not permit tables `DISTRIBUTED RANDOMLY` to have primary keys or unique indexes. Restoring such a table from a Greenplum 4.3 or 5 backup will cause an error.
- Greenplum 6 no longer automatically converts from the deprecated timestamp format `YYYYMMDDHH24MISS`. The format could not be parsed unambiguously in previous Greenplum Database releases. You can still specify the `YYYYMMDDHH24MISS` format in conversion functions such

as `to_timestamp` and `to_char` for compatibility with other database systems. You can use input formats for converting text to date or timestamp values to avoid unexpected results or query execution failures. For example, this `SELECT` command returns a timestamp in Greenplum Database 5 and fails in 6.

```
SELECT to_timestamp('20190905140000');
```

To convert the string to a timestamp in Greenplum Database 6, you must use a valid format. Both of these commands return a timestamp in Greenplum Database 6. The first example explicitly specifies a timestamp format. The second example uses the string in a format that Greenplum Database recognizes.

```
SELECT to_timestamp('20190905140000','YYYYMMDDHH24MISS');
SELECT to_timestamp('201909051 40000');
```

The timestamp issue also applies when you use the `::` syntax. In Greenplum Database 6, the first command returns an error. The second command returns a timestamp.

```
SELECT '20190905140000'::timestamp ;
SELECT '20190905 140000'::timestamp ;
```

- Creating a table using the `CREATE TABLE AS` command in Greenplum 4.3 or 5 could create a table with a duplicate distribution key. The `gpbackup` utility saves the table to the backup using a `CREATE TABLE` command that lists the duplicate keys in the `DISTRIBUTED BY` clause. Restoring this backup will cause a duplicate distribution key error. The `CREATE TABLE AS` command was fixed in Greenplum 5.10 to disallow duplicate distribution keys.
- Greenplum 4.3 supports foreign key constraints on columns of different types, for example, `numeric` and `bigint`, with implicit type conversion. Greenplum 5 and 6 do not support implicit type conversion. Restoring a table with a foreign key on columns with different data types causes an error.
- Only Boolean operators can use Boolean negators. In Greenplum Database 4.3 and 5 it was possible to create a non-Boolean operator that specifies a Boolean negator function. For example, this `CREATE OPERATOR` command creates an integer `@@` operator with a Boolean negator:

  ```
  CREATE OPERATOR public.@@ (
          PROCEDURE = int4pl,
          LEFTARG = integer,
          RIGHTARG = integer,
          NEGATOR = OPERATOR(public.!!)
  );
  ```

  If you restore a backup containing an operator like this to a Greenplum 6 system, `gprestore` produces an error: `ERROR: only boolean operators can have negators (SQLSTATE 42P13)`.
- In Greenplum Database 4.3 and 5, the undocumented server configuration parameter `allow_system_table_mods` could have a value of `none`, `ddl`, `dml`, or `all`. In Greenplum 6, this parameter has changed to a Boolean value, with a default value of `false`. If there are any references to this parameter in the source database, remove them to prevent errors during the restore.

Backing Up and Restoring a Database

First use `gpbackup` to create a `--metadata-only` backup from the source Greenplum database and restore it to the Greenplum 6 system. This helps find any additional problems that are not identified in *Preparing Greenplum 4.3 and 5 Databases for Backup*. Refer to the *Greenplum Backup and Restore documentation* for syntax and examples for the `gpbackup` and `gprestore` utilities.

Review the `gprestore` log file for error messages and correct any remaining problems in the source Greenplum database.

When you are able to restore a metadata backup successfully, create the full backup and then restore it to the Greenplum 6 system, or use `gpcopy` to transfer the data. If needed, use the `gpbackup` or `gprestore` filter options to omit schemas or tables that cannot be restored without error.

If you use `gpcopy` to migrate data, initiate the `gpcopy` operation from the Greenplum 4.3.26 (or later) or the 5.9 (or later) cluster. See *Migrating Data with gpcopy* for more information.

> **Important:** When you restore a backup taken from a Greenplum Database 4.3 or 5 system, `gprestore` warns that the restore will use legacy hash operators when loading the data. This is because Greenplum 6 has new hash algorithms that map distribution keys to segments, but the data in the backup set must be restored to the same segments as the cluster from which the backup was taken. The `gprestore` utility sets the `gp_use_legacy_hashops` server configuration parameter to `on` when restoring to Greenplum 6 from an earlier version so that the restored tables are created using the legacy operator classes instead of the new default operator classes.
>
> After restoring, you can redistribute these tables with the `gp_use_legacy_hashops` parameter set to `off` so that the tables use the new Greenplum 6 hash operators. See *Working With Hash Operator Classes in Greenplum 6* for more information and examples.

Completing the Migration

Migrate any tables you skipped during the restore using other methods, for example using the `COPY TO` command to create an external file and then loading the data from the external file into Greenplum 6 with the `COPY FROM` command.

Recreate any objects you dropped in the Greenplum 4.3 or 5 database to enable migration, such as external tables, indexes, user-defined functions, or user-defined aggregates.

Here are some additional items to consider to complete your migration to Greenplum 6.

- If you are migrating from Greenplum Database 4.3.27 or an earlier 4.3.x release and have configured PgBouncer in your Greenplum Database installation, you must migrate to the new PgBouncer when you upgrade Greenplum Database. Refer to *Migrating PgBouncer* for specific migration instructions.
- Greenplum Database 5 and 6 remove automatic casts between the text type and other data types. After you migrate from Greenplum Database version 4.3 to version 6, this changed behavior may impact existing applications and queries. Refer to *About Implicit Text Casting in Greenplum Database* for information, including a discussion about supported and unsupported workarounds.
- After migrating data you may need to modify SQL scripts, administration scripts, and user-defined functions as necessary to account for changes in Greenplum Database version 6. Review the *Pivotal Greenplum 6.0.0 Release Notes* for features and changes that may necessitate post-migration tasks.
- To use the new Greenplum 6 default hash operator classes, use the command `ALTER TABLE <table> SET DISTRIBUTED BY (<key>)` to redistribute tables restored from Greenplum 4.3 or 5 backups. The `gp_use_legacy_hashops` parameter must be set to `off` when your execute the command. See *Working With Hash Operator Classes in Greenplum 6* for more information about hash operator classes.

Working With Hash Operator Classes in Greenplum 6

Greenplum 6 has new *jump consistent* hash operators that map distribution keys for distributed tables to the segments. The new hash operators enable faster database expansion because they don't require redistributing rows unless they map to a different segment. The hash operators used in Greenplum 4.3 and 5 are present in Greenplum 6 as non-default legacy hash operator classes. For example, for integer columns, the new hash operator class is named `int_ops` and the legacy operator class is named `cdbhash_int_ops`.

This example creates a table using the legacy hash operator class `cdbhash_int_ops`.

```
test=# SET gp_use_legacy_hashops=on;
SET
```

```
test=# CREATE TABLE t1 (
    c1 integer,
    c2 integer,
    p integer
) DISTRIBUTED BY (c1);
CREATE TABLE
test=# \d+ t1
                          Table "public.t1"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 c1     | integer |           | plain   |              |
 c2     | integer |           | plain   |              |
 p      | integer |           | plain   |              |
Distributed by: (c1)
```

Notice that the distribution key is `c1`. If the `gp_use_legacy_hashops` parameter is `on` and the operator class is a legacy operator class, the operator class name is not shown. However, if `gp_use_legacy_hashops` is `off`, the legacy operator class name is reported with the distribution key.

```
test=# SET gp_use_legacy_hashops=off;
SET
test=# \d+ t1
                          Table "public.t1"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 c1     | integer |           | plain   |              |
 c2     | integer |           | plain   |              |
 p      | integer |           | plain   |              |
Distributed by: (c1 cdbhash_int4_ops)
```

The operator class name is reported only when it does not match the setting of the `gp_use_legacy_hashops` parameter.

To change the table to use the new jump consistent operator class, use the `ALTER TABLE` command to redistribute the table with the `gp_use_legacy_hashops` parameter set to `off`.

Note: Redistributing tables with a large amount of data can take a long time.

```
test=# SHOW gp_use_legacy_hashops;
 gp_use_legacy_hashops
-----------------------
 off
(1 row)

test=# ALTER TABLE t1 SET DISTRIBUTED BY (c1);
ALTER TABLE
test=# \d+ t1
                          Table "public.t1"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 c1     | integer |           | plain   |              |
 c2     | integer |           | plain   |              |
 p      | integer |           | plain   |              |
Distributed by: (c1)
```

To verify the default jump consistent operator class has been used, set `gp_use_legacy_hashops` to on before you show the table definition.

```
test=# SET gp_use_legacy_hashops=on;
SET
test=# \d+ t1
                          Table "public.t1"
```

```
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 c1     | integer |           | plain   |              |
 c2     | integer |           | plain   |              |
 p      | integer |           | plain   |              |
Distributed by: (c1 int4_ops)
```

Enabling iptables (Optional)

On Linux systems, you can configure and enable the `iptables` firewall to work with Greenplum Database.

> **Note:** Greenplum Database performance might be impacted when `iptables` is enabled. You should test the performance of your application with `iptables` enabled to ensure that performance is acceptable.

For more information about `iptables` see the `iptables` and firewall documentation for your operating system. See also *Disabling SELinux and Firewall Software*.

How to Enable iptables

1. As `gpadmin`, run this command on the Greenplum Database master host to stop Greenplum Database:

```
$ gpstop -a
```

2. On the Greenplum Database hosts:
   a. Update the file `/etc/sysconfig/iptables` based on the *Example iptables Rules*.
   b. As root user, run these commands to enable `iptables`:

```
# chkconfig iptables on
# service iptables start
```

3. As gpadmin, run this command on the Greenplum Database master host to start Greenplum Database:

```
$ gpstart -a
```

> **Warning:** After enabling `iptables`, this error in the `/var/log/messages` file indicates that the setting for the `iptables` table is too low and needs to be increased.

```
ip_conntrack: table full, dropping packet.
```

As root, run this command to view the `iptables` table value:

```
# sysctl net.ipv4.netfilter.ip_conntrack_max
```

To ensure that the Greenplum Database workload does not overflow the `iptables` table, as root, set it to the following value:

```
# sysctl net.ipv4.netfilter.ip_conntrack_max=6553600
```

The value might need to be adjusted for your hosts. To maintain the value after reboot, you can update the `/etc/sysctl.conf` file as discussed in *Setting the Greenplum Recommended OS Parameters*.

Example iptables Rules

When `iptables` is enabled, `iptables` manages the IP communication on the host system based on configuration settings (rules). The example rules are used to configure `iptables` for Greenplum Database master host, standby master host, and segment hosts.

- *Example Master and Standby Master iptables Rules*
- *Example Segment Host iptables Rules*

The two sets of rules account for the different types of communication Greenplum Database expects on the master (primary and standby) and segment hosts. The rules should be added to the `/etc/sysconfig/iptables` file of the Greenplum Database hosts. For Greenplum Database, `iptables` rules should allow the following communication:

- For customer facing communication with the Greenplum Database master, allow at least `postgres` and `28080` (`eth1` interface in the example).
- For Greenplum Database system interconnect, allow communication using `tcp`, `udp`, and `icmp` protocols (`eth4` and `eth5` interfaces in the example).

  The network interfaces that you specify in the `iptables` settings are the interfaces for the Greenplum Database hosts that you list in the *hostfile_gpinitsystem* file. You specify the file when you run the `gpinitsystem` command to initialize a Greenplum Database system. See *Initializing a Greenplum Database System* for information about the *hostfile_gpinitsystem* file and the `gpinitsystem` command.
- For the administration network on a Greenplum DCA, allow communication using `ssh`, `ntp`, and `icmp` protocols. (`eth0` interface in the example).

In the `iptables` file, each append rule command (lines starting with `-A`) is a single line.

The example rules should be adjusted for your configuration. For example:

- The append command, the `-A` lines and connection parameter `-i` should match the connectors for your hosts.
- the CIDR network mask information for the source parameter `-s` should match the IP addresses for your network.

Example Master and Standby Master iptables Rules

Example `iptables` rules with comments for the `/etc/sysconfig/iptables` file on the Greenplum Database master host and standby master host.

```
*filter
# Following 3 are default rules. If the packet passes through
# the rule set it gets these rule.
# Drop all inbound packets by default.
# Drop all forwarded (routed) packets.
# Let anything outbound go through.
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
# Accept anything on the loopback interface.
-A INPUT -i lo -j ACCEPT
# If a connection has already been established allow the
# remote host packets for the connection to pass through.
-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
# These rules let all tcp and udp through on the standard
# interconnect IP addresses and on the interconnect interfaces.
# NOTE: gpsyncmaster uses random tcp ports in the range 1025 to 65535
# and Greenplum Database uses random udp ports in the range 1025 to 65535.
-A INPUT -i eth4 -p udp -s 192.0.2.0/22 -j ACCEPT
-A INPUT -i eth5 -p udp -s 198.51.100.0/22 -j ACCEPT
-A INPUT -i eth4 -p tcp -s 192.0.2.0/22 -j ACCEPT --syn -m state --state NEW
-A INPUT -i eth5 -p tcp -s 198.51.100.0/22 -j ACCEPT --syn -m state --state
 NEW
# Allow udp/tcp ntp connections on the admin network on Greenplum DCA.
-A INPUT -i eth0 -p udp --dport ntp -s 203.0.113.0/21 -j ACCEPT
-A INPUT -i eth0 -p tcp --dport ntp -s 203.0.113.0/21 -j ACCEPT --syn -m
 state --state NEW
# Allow ssh on all networks (This rule can be more strict).
-A INPUT -p tcp --dport ssh -j ACCEPT --syn -m state --state NEW
# Allow Greenplum Database on all networks.
```

```
-A INPUT -p tcp --dport postgres -j ACCEPT --syn -m state --state NEW
# Allow Greenplum Command Center on the customer facing network.
-A INPUT -i eth1 -p tcp --dport 28080 -j ACCEPT --syn -m state --state NEW
# Allow ping and any other icmp traffic on the interconnect networks.
-A INPUT -i eth4 -p icmp -s 192.0.2.0/22 -j ACCEPT
-A INPUT -i eth5 -p icmp -s 198.51.100.0/22 -j ACCEPT
# Allow ping only on the admin network on Greenplum DCA.
-A INPUT -i eth0 -p icmp --icmp-type echo-request -s 203.0.113.0/21 -j
 ACCEPT
# Log an error if a packet passes through the rules to the default
# INPUT rule (a DROP).
-A INPUT -m limit --limit 5/min -j LOG --log-prefix "iptables denied: " --
log-level 7
COMMIT
```

Example Segment Host iptables Rules

Example `iptables` rules for the `/etc/sysconfig/iptables` file on the Greenplum Database segment hosts. The rules for segment hosts are similar to the master rules with fewer interfaces and fewer `udp` and `tcp` services.

```
*filter
:INPUT DROP
:FORWARD DROP
:OUTPUT ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
-A INPUT -i eth2 -p udp -s 192.0.2.0/22 -j ACCEPT
-A INPUT -i eth3 -p udp -s 198.51.100.0/22 -j ACCEPT
-A INPUT -i eth2 -p tcp -s 192.0.2.0/22 -j ACCEPT --syn -m state --state NEW
-A INPUT -i eth3 -p tcp -s 198.51.100.0/22 -j ACCEPT --syn -m state --state
 NEW
-A INPUT -p tcp --dport ssh -j ACCEPT --syn -m state --state NEW
-A INPUT -i eth2 -p icmp -s 192.0.2.0/22 -j ACCEPT
-A INPUT -i eth3 -p icmp -s 198.51.100.0/22 -j ACCEPT
-A INPUT -i eth0 -p icmp --icmp-type echo-request -s 203.0.113.0/21 -j
 ACCEPT
-A INPUT -m limit --limit 5/min -j LOG --log-prefix "iptables denied: " --
log-level 7
COMMIT
```

Installation Management Utilities

References for the command-line management utilities used to install and initialize a Greenplum Database system.

For a full reference of all Greenplum Database utilities, see the *Greenplum Database Utility Guide*.

The following Greenplum Database management utilities are located in `$GPHOME/bin`.

• *gpactivatestandby* • *gpaddmirrors* • *gpcheckperf* • *gpcopy* • *gpdeletesystem* • *gpinitstandby*	• *gpinitsystem* • *gppkg* • *gpscp* • *gpssh* • *gpssh-exkeys* • *gpstart* • *gpstop*

Greenplum Environment Variables

Reference of the environment variables to set for Greenplum Database.

Set these in your user's startup shell profile (such as `~/.bashrc` or `~/.bash_profile`), or in `/etc/profile` if you want to set them for all users.

Required Environment Variables

Note: `GPHOME`, `PATH` and `LD_LIBRARY_PATH` can be set by sourcing the `greenplum_path.sh` file from your Greenplum Database installation directory

GPHOME

This is the installed location of your Greenplum Database software. For example:

```
GPHOME=/usr/local/greenplum-db-6.x.x
export GPHOME
```

PATH

Your `PATH` environment variable should point to the location of the Greenplum Database `bin` directory. For example:

```
PATH=$GPHOME/bin:$PATH
export PATH
```

LD_LIBRARY_PATH

The `LD_LIBRARY_PATH` environment variable should point to the location of the Greenplum Database/PostgreSQL library files. For example:

```
LD_LIBRARY_PATH=$GPHOME/lib
export LD_LIBRARY_PATH
```

MASTER_DATA_DIRECTORY

This should point to the directory created by the gpinitsystem utility in the master data directory location. For example:

```
MASTER_DATA_DIRECTORY=/data/master/gpseg-1
export MASTER_DATA_DIRECTORY
```

Optional Environment Variables

The following are standard PostgreSQL environment variables, which are also recognized in Greenplum Database. You may want to add the connection-related environment variables to your profile for convenience, so you do not have to type so many options on the command line for client connections. Note that these environment variables should be set on the Greenplum Database master host only.

PGAPPNAME

The name of the application that is usually set by an application when it connects to the server. This name is displayed in the activity view and in log entries. The `PGAPPNAME` environmental variable behaves the same as the `application_name` connection parameter. The default value for `application_name` is `psql`. The name cannot be longer than 63 characters.

PGDATABASE

The name of the default database to use when connecting.

PGHOST

The Greenplum Database master host name.

PGHOSTADDR

The numeric IP address of the master host. This can be set instead of or in addition to `PGHOST` to avoid DNS lookup overhead.

PGPASSWORD

The password used if the server demands password authentication. Use of this environment variable is not recommended for security reasons (some operating systems allow non-root users to see process environment variables via `ps`). Instead consider using the `~/.pgpass` file.

PGPASSFILE

The name of the password file to use for lookups. If not set, it defaults to `~/.pgpass`. See the topic about *The Password File* in the PostgreSQL documentation for more information.

PGOPTIONS

Sets additional configuration parameters for the Greenplum Database master server.

PGPORT

The port number of the Greenplum Database server on the master host. The default port is 5432.

PGUSER

The Greenplum Database user name used to connect.

PGDATESTYLE

Sets the default style of date/time representation for a session. (Equivalent to `SET datestyle TO...`)

PGTZ

Sets the default time zone for a session. (Equivalent to `SET timezone TO...`)

PGCLIENTENCODING

Sets the default client character set encoding for a session. (Equivalent to `SET client_encoding TO...`)

Example Ansible Playbook

A sample Ansible playbook to install a Greenplum Database software release onto the hosts that will comprise a Greenplum Database system.

This Ansible playbook shows how tasks described in *Installing the Greenplum Database Software* might be automated using *Ansible*.

> **Important:** This playbook is provided as an *example only* to illustrate how Greenplum Database cluster configuration and software installation tasks can be automated using provisioning tools such as Ansible, Chef, or Puppet. Pivotal does not provide support for Ansible or for the playbook presented in this example.

The example playbook is designed for use with CentOS 7. It creates the `gpadmin` user, installs the Greenplum Database software release, sets the owner and group of the installed software to `gpadmin`, and sets the Pam security limits for the `gpadmin` user.

You can revise the script to work with your operating system platform and to perform additional host configuration tasks.

Following are steps to use this Ansible playbook.

1. Install Ansible on the control node using your package manager. See the *Ansible documentation* for help with installation.
2. Set up passwordless SSH from the control node to all hosts that will be a part of the Greenplum Database cluster. You can use the `ssh-copy-id` command to install your public SSH key on each host in the cluster. Alternatively, your provisioning software may provide more convenient ways to securely install public keys on multiple hosts.
3. Create an Ansible inventory by creating a file called `hosts` with a list of the hosts that will comprise your Greenplum Database cluster. For example:

   ```
   mdw
   sdw1
   sdw2
   ...
   ```

   This file can be edited and used with the Greenplum Database `gpssh-exkeys` and `gpinitsystem` utilities later on.
4. Copy the playbook code below to a file `ansible-playbook.yml` on your Ansible control node.
5. Edit the playbook variables at the top of the playbook, such as the `gpadmin` administrative user and password to create, and the version of Greenplum Database you are installing.
6. Run the playbook, passing the package to be installed to the `package_path` parameter.

   ```
   ansible-playbook ansible-playbook.yml -i hosts -e package_path=./
   greenplum-db-6.0.0-rhel7-x86_64.rpm
   ```

Ansible Playbook - Greenplum Database Installation for CentOS 7

```
---

- hosts: all
  vars:
    - version: "6.0.0"
    - greenplum_admin_user: "gpadmin"
    - greenplum_admin_password: "changeme"
```

```
    # - package_path: passed via the command line with: -e package_path=./
greenplum-db-6.0.0-rhel7-x86_64.rpm
  remote_user: root
  become: yes
  become_method: sudo
  connection: ssh
  gather_facts: yes
  tasks:
    - name: create greenplum admin user
      user:
        name: "{{ greenplum_admin_user }}"
        password: "{{ greenplum_admin_password | password_hash('sha512',
 'DvkPtCtNH+UdbePZfm9muQ9pU') }}"
    - name: copy package to host
      copy:
        src: "{{ package_path }}"
        dest: /tmp
    - name: install package
      yum:
        name: "/tmp/{{ package_path | basename }}"
        state: present
    - name: cleanup package file from host
      file:
        path: "/tmp/{{ package_path | basename }}"
        state: absent
    - name: find install directory
      find:
        paths: /usr/local
        patterns: 'greenplum*'
        file_type: directory
      register: installed_dir
    - name: change install directory ownership
      file:
        path: '{{ item.path }}'
        owner: "{{ greenplum_admin_user }}"
        group: "{{ greenplum_admin_user }}"
        recurse: yes
      with_items: "{{ installed_dir.files }}"
    - name: update pam_limits
      pam_limits:
        domain: "{{ greenplum_admin_user }}"
        limit_type: '-'
        limit_item: "{{ item.key }}"
        value: "{{ item.value }}"
      with_dict:
        nofile: 524288
        nproc: 131072
    - name: find installed greenplum version
      shell: . /usr/local/greenplum-db/greenplum_path.sh && /usr/local/
greenplum-db/bin/postgres --gp-version
      register: postgres_gp_version
    - name: fail if the correct greenplum version is not installed
      fail:
        msg: "Expected greenplum version {{ version }}, but found
 '{{ postgres_gp_version.stdout }}'"
      when: "version is not defined or version not in
 postgres_gp_version.stdout"
```

When the playbook has executed successfully, you can proceed with *Creating the Data Storage Areas* and *Initializing a Greenplum Database System*.

Chapter 3

Greenplum Database Administrator Guide

Information about configuring, managing and monitoring Greenplum Database installations, and administering, monitoring, and working with databases. The guide also contains information about Greenplum Database architecture and concepts such as parallel processing.

Greenplum Database Concepts

This section provides an overview of Greenplum Database components and features such as high availability, parallel data loading features, and management utilities.

This section contains the following topics:

About the Greenplum Architecture

Greenplum Database is a massively parallel processing (MPP) database server with an architecture specially designed to manage large-scale analytic data warehouses and business intelligence workloads.

MPP (also known as a *shared nothing* architecture) refers to systems with two or more processors that cooperate to carry out an operation, each processor with its own memory, operating system and disks. Greenplum uses this high-performance system architecture to distribute the load of multi-terabyte data warehouses, and can use all of a system's resources in parallel to process a query.

Greenplum Database is based on PostgreSQL open-source technology. It is essentially several PostgreSQL disk-oriented database instances acting together as one cohesive database management system (DBMS). It is based on PostgreSQL 9.4, and in most cases is very similar to PostgreSQL with regard to SQL support, features, configuration options, and end-user functionality. Database users interact with Greenplum Database as they would with a regular PostgreSQL DBMS.

Greenplum Database can use the append-optimized (AO) storage format for bulk loading and reading of data, and provides performance advantages over HEAP tables. Append-optimized storage provides checksums for data protection, compression and row/column orientation. Both row-oriented or column-oriented append-optimized tables can be compressed.

The main differences between Greenplum Database and PostgreSQL are as follows:

- GPORCA is leveraged for query planning, in addition to the Postgres Planner.
- Greenplum Database can use append-optimized storage.
- Greenplum Database has the option to use column storage, data that is logically organized as a table, using rows and columns that are physically stored in a column-oriented format, rather than as rows. Column storage can only be used with append-optimized tables. Column storage is compressible. It also can provide performance improvements as you only need to return the columns of interest to you. All compression algorithms can be used with either row or column-oriented tables, but Run-Length Encoded (RLE) compression can only be used with column-oriented tables. Greenplum Database provides compression on all Append-Optimized tables that use column storage.

The internals of PostgreSQL have been modified or supplemented to support the parallel structure of Greenplum Database. For example, the system catalog, optimizer, query executor, and transaction manager components have been modified and enhanced to be able to execute queries simultaneously across all of the parallel PostgreSQL database instances. The Greenplum *interconnect* (the networking layer) enables communication between the distinct PostgreSQL instances and allows the system to behave as one logical database.

Greenplum Database also can use declarative partitions and sub-partitions to implicitly generate partition constraints.

Greenplum Database also includes features designed to optimize PostgreSQL for business intelligence (BI) workloads. For example, Greenplum has added parallel data loading (external tables), resource

management, query optimizations, and storage enhancements, which are not found in standard PostgreSQL. Many features and optimizations developed by Greenplum make their way into the PostgreSQL community. For example, table partitioning is a feature first developed by Greenplum, and it is now in standard PostgreSQL.

Greenplum Database queries use a Volcano-style query engine model, where the execution engine takes an execution plan and uses it to generate a tree of physical operators, evaluates tables through physical operators, and delivers results in a query response.

Greenplum Database stores and processes large amounts of data by distributing the data and processing workload across several servers or *hosts*. Greenplum Database is an *array* of individual databases based upon PostgreSQL 9.4 working together to present a single database image. The *master* is the entry point to the Greenplum Database system. It is the database instance to which clients connect and submit SQL statements. The master coordinates its work with the other database instances in the system, called *segments*, which store and process the data.

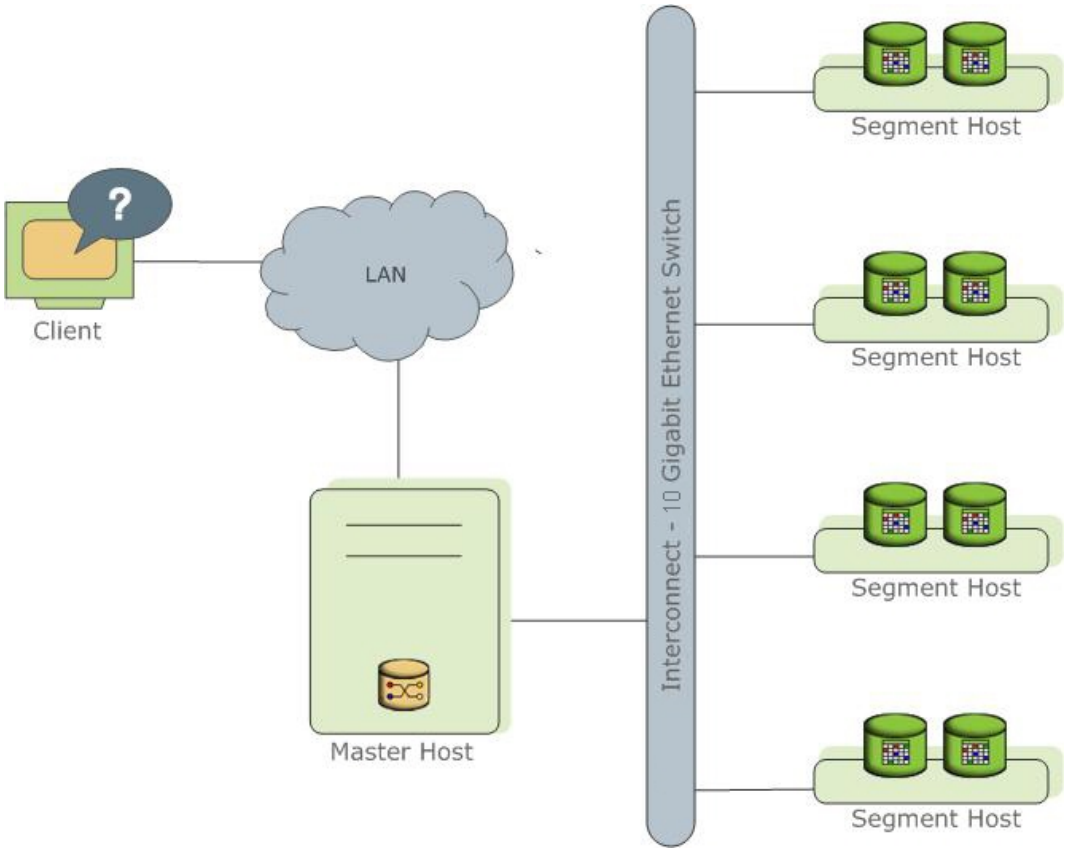


Figure 10: High-Level Greenplum Database Architecture

The following topics describe the components that make up a Greenplum Database system and how they work together.

About the Greenplum Master

The Greenplum Database master is the entry to the Greenplum Database system, accepting client connections and SQL queries, and distributing work to the segment instances.

Greenplum Database end-users interact with Greenplum Database (through the master) as they would with a typical PostgreSQL database. They connect to the database using client programs such as `psql` or application programming interfaces (APIs) such as JDBC, ODBC or *libpq* (the PostgreSQL C API).

The master is where the *global system catalog* resides. The global system catalog is the set of system tables that contain metadata about the Greenplum Database system itself. The master does not contain

any user data; data resides only on the *segments*. The master authenticates client connections, processes incoming SQL commands, distributes workloads among segments, coordinates the results returned by each segment, and presents the final results to the client program.

Greenplum Database uses Write-Ahead Logging (WAL) for master/standby master mirroring. In WAL-based logging, all modifications are written to the log before being applied, to ensure data integrity for any in-process operations.

About the Greenplum Segments

Greenplum Database segment instances are independent PostgreSQL databases that each store a portion of the data and perform the majority of query processing.

When a user connects to the database via the Greenplum master and issues a query, processes are created in each segment database to handle the work of that query. For more information about query processes, see *About Greenplum Query Processing*.

User-defined tables and their indexes are distributed across the available segments in a Greenplum Database system; each segment contains a distinct portion of data. The database server processes that serve segment data run under the corresponding segment instances. Users interact with segments in a Greenplum Database system through the master.

Segments run on a servers called *segment hosts*. A segment host typically executes from two to eight Greenplum segments, depending on the CPU cores, RAM, storage, network interfaces, and workloads. Segment hosts are expected to be identically configured. The key to obtaining the best performance from Greenplum Database is to distribute data and workloads *evenly* across a large number of equally capable segments so that all segments begin working on a task simultaneously and complete their work at the same time.

About the Greenplum Interconnect

The interconnect is the networking layer of the Greenplum Database architecture.

The *interconnect* refers to the inter-process communication between segments and the network infrastructure on which this communication relies. The Greenplum interconnect uses a standard Ethernet switching fabric. For performance reasons, a 10-Gigabit system, or faster, is recommended.

By default, the interconnect uses User Datagram Protocol with flow control (UDPIFC) for interconnect traffic to send messages over the network. The Greenplum software performs packet verification beyond what is provided by UDP. This means the reliability is equivalent to Transmission Control Protocol (TCP), and the performance and scalability exceeds TCP. If the interconnect is changed to TCP, Greenplum Database has a scalability limit of 1000 segment instances. With UDPIFC as the default protocol for the interconnect, this limit is not applicable.

About Management and Monitoring Utilities

Greenplum Database provides standard command-line utilities for performing common monitoring and administration tasks.

Greenplum command-line utilities are located in the *$GPHOME*`/bin` directory and are executed on the master host. Greenplum provides utilities for the following administration tasks:

- Installing Greenplum Database on an array
- Initializing a Greenplum Database System
- Starting and stopping Greenplum Database
- Adding or removing a host
- Expanding the array and redistributing tables among new segments
- Managing recovery for failed segment instances
- Managing failover and recovery for a failed master instance
- Backing up and restoring a database (in parallel)

- Loading data in parallel
- Transferring data between Greenplum databases
- System state reporting

Greenplum Database includes an optional performance management database that contains query status information and system metrics. The `gpperfmon_install` management utility creates the database, named `gpperfmon`, and enables data collection agents that execute on the Greenplum Database master and segment hosts. Data collection agents on the segment hosts collect query status from the segments, as well as system metrics such as CPU and memory utilization. An agent on the master host periodically (typically every 15 seconds) retrieves the data from the segment host agents and updates the `gpperfmon` database. Users can query the `gpperfmon` database to see the query and system metrics.

Pivotal provides an optional system monitoring and management tool, Greenplum Command Center, which administrators can install and enable with Greenplum Database. Greenplum Command Center provides a web-based user interface for viewing system metrics and allows administrators to perform additional system management tasks. For more information about Greenplum Command Center, see the *Greenplum Command Center documentation*.

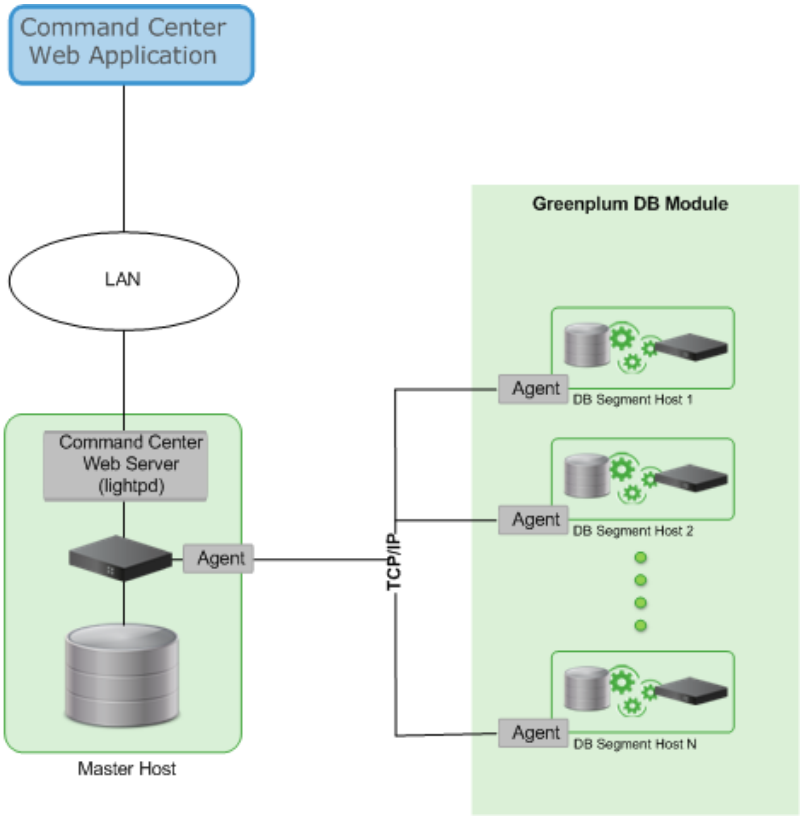


Figure 11: Greenplum Command Center Architecture

About Concurrency Control in Greenplum Database

Greenplum Database uses the PostgreSQL Multiversion Concurrency Control (MVCC) model to manage concurrent transactions for heap tables.

Concurrency control in a database management system allows concurrent queries to complete with correct results while ensuring the integrity of the database. Traditional databases use a two-phase locking protocol that prevents a transaction from modifying data that has been read by another concurrent transaction and prevents any concurrent transaction from reading or writing data that another transaction has updated. The locks required to coordinate transactions add contention to the database, reducing overall transaction throughput.

Greenplum Database uses the PostgreSQL Multiversion Concurrency Control (MVCC) model to manage concurrency for heap tables. With MVCC, each query operates on a snapshot of the database when the query starts. While it executes, a query cannot see changes made by other concurrent transactions. This ensures that a query sees a consistent view of the database. Queries that read rows can never block waiting for transactions that write rows. Conversely, queries that write rows cannot be blocked by transactions that read rows. This allows much greater concurrency than traditional database systems that employ locks to coordinate access between transactions that read and write data.

> **Note:** Append-optimized tables are managed with a different concurrency control model than the MVCC model discussed in this topic. They are intended for "write-once, read-many" applications that never, or only very rarely, perform row-level updates.

Snapshots

The MVCC model depends on the system's ability to manage multiple versions of data rows. A query operates on a snapshot of the database at the start of the query. A snapshot is the set of rows that are visible at the beginning of a statement or transaction. The snapshot ensures the query has a consistent and valid view of the database for the duration of its execution.

Each transaction is assigned a unique *transaction ID* (XID), an incrementing 32-bit value. When a new transaction starts, it is assigned the next XID. An SQL statement that is not enclosed in a transaction is treated as a single-statement transaction—the `BEGIN` and `COMMIT` are added implicitly. This is similar to autocommit in some database systems.

> **Note:** Greenplum Database assigns XID values only to transactions that involve DDL or DML operations, which are typically the only transactions that require an XID.

When a transaction inserts a row, the XID is saved with the row in the `xmin` system column. When a transaction deletes a row, the XID is saved in the `xmax` system column. Updating a row is treated as a delete and an insert, so the XID is saved to the `xmax` of the current row and the `xmin` of the newly inserted row. The `xmin` and `xmax` columns, together with the transaction completion status, specify a range of transactions for which the version of the row is visible. A transaction can see the effects of all transactions less than `xmin`, which are guaranteed to be committed, but it cannot see the effects of any transaction greater than or equal to `xmax`.

Multi-statement transactions must also record which command within a transaction inserted a row (`cmin`) or deleted a row (`cmax`) so that the transaction can see changes made by previous commands in the transaction. The command sequence is only relevant during the transaction, so the sequence is reset to 0 at the beginning of a transaction.

XID is a property of the database. Each segment database has its own XID sequence that cannot be compared to the XIDs of other segment databases. The master coordinates distributed transactions with the segments using a cluster-wide *session ID number*, called `gp_session_id`. The segments maintain a mapping of distributed transaction IDs with their local XIDs. The master coordinates distributed transactions across all of the segment with the two-phase commit protocol. If a transaction fails on any one segment, it is rolled back on all segments.

You can see the `xmin`, `xmax`, `cmin`, and `cmax` columns for any row with a `SELECT` statement:

```
SELECT xmin, xmax, cmin, cmax, * FROM tablename;
```

Because you run the `SELECT` command on the master, the XIDs are the distributed transactions IDs. If you could execute the command in an individual segment database, the `xmin` and `xmax` values would be the segment's local XIDs.

> **Note:** Greenplum Database distributes all of a replicated table's rows to every segment, so each row is duplicated on every segment. Each segment instance maintains its own values for the system columns `xmin`, `xmax`, `cmin`, and `cmax`, as well as for the `gp_segment_id` and `ctid`system columns. Greenplum Database does not permit user queries to access these system

columns for replicated tables because they have no single, unambiguous value to evaluate in a query.

Transaction ID Wraparound

The MVCC model uses transaction IDs (XIDs) to determine which rows are visible at the beginning of a query or transaction. The XID is a 32-bit value, so a database could theoretically execute over four billion transactions before the value overflows and wraps to zero. However, Greenplum Database uses *modulo* 2^{32} arithmetic with XIDs, which allows the transaction IDs to wrap around, much as a clock wraps at twelve o'clock. For any given XID, there could be about two billion past XIDs and two billion future XIDs. This works until a version of a row persists through about two billion transactions, when it suddenly appears to be a new row. To prevent this, Greenplum has a special XID, called `FrozenXID`, which is always considered older than any regular XID it is compared with. The `xmin` of a row must be replaced with `FrozenXID` within two billion transactions, and this is one of the functions the `VACUUM` command performs.

Vacuuming the database at least every two billion transactions prevents XID wraparound. Greenplum Database monitors the transaction ID and warns if a `VACUUM` operation is required.

A warning is issued when a significant portion of the transaction IDs are no longer available and before transaction ID wraparound occurs:

```
WARNING: database "database_name" must be vacuumed
 within number_of_transactions transactions
```

When the warning is issued, a `VACUUM` operation is required. If a `VACUUM` operation is not performed, Greenplum Database stops creating transactions to avoid possible data loss when it reaches a limit prior to when transaction ID wraparound occurs and issues this error:

```
FATAL: database is not accepting commands to avoid wraparound data loss in
 database "database_name"
```

See *Recovering from a Transaction ID Limit Error* for the procedure to recover from this error.

The server configuration parameters `xid_warn_limit` and `xid_stop_limit` control when the warning and error are displayed. The `xid_warn_limit` parameter specifies the number of transaction IDs before the `xid_stop_limit` when the warning is issued. The `xid_stop_limit` parameter specifies the number of transaction IDs before wraparound would occur when the error is issued and new transactions cannot be created.

Transaction Isolation Modes

The SQL standard describes three phenomena that can occur when database transactions run concurrently:

- *Dirty read* – a transaction can read uncommitted data from another concurrent transaction.
- *Non-repeatable read* – a row read twice in a transaction can change because another concurrent transaction committed changes after the transaction began.
- *Phantom read* – a query executed twice in the same transaction can return two different sets of rows because another concurrent transaction added rows.

The SQL standard defines four transaction isolation levels that database systems can support, with the phenomena that are allowed when transactions execute concurrently for each level.

Table 11: SQL Transaction Isolation Modes

Level	Dirty Read	Non-Repeatable	Phantom Read
Read Uncommitted	Possible	Possible	Possible
Read Committed	Impossible	Possible	Possible

Level	Dirty Read	Non-Repeatable	Phantom Read
Repeatable Read	Impossible	Impossible	Possible
Serializable	Impossible	Impossible	Impossible

Greenplum Database `READ UNCOMMITTED` and `READ COMMITTED` isolation modes behave like the SQL standard `READ COMMITTED` mode. Greenplum Database `SERIALIZABLE` and `REPEATABLE READ` isolation modes behave like the SQL standard `READ COMMITTED` mode, except that Greenplum Database also prevents phantom reads.

The difference between `READ COMMITTED` and `REPEATABLE READ` is that with `READ COMMITTED`, each statement in a transaction sees only rows committed before the *statement* started, while in `READ COMMITTED` mode, statements in a transaction see only rows committed before the *transaction* started.

With `READ COMMITTED` isolation mode the values in a row retrieved twice in a transaction can differ if another concurrent transaction has committed changes since the transaction began. `READ COMMITTED` mode also permits *phantom reads*, where a query executed twice in the same transaction can return two different sets of rows.

The `REPEATABLE READ` isolation mode prevents non-repeatable reads and phantom reads, although the latter is not required by the standard. A transaction that attempts to modify data modified by another concurrent transaction is rolled back. Applications that execute transactions in `REPEATABLE READ` mode must be prepared to handle transactions that fail due to serialization errors. If `REPEATABLE READ` isolation mode is not required by the application, it is better to use `READ COMMITTED` mode.

`SERIALIZABLE` mode, which Greenplum Database does not fully support, guarantees that a set of transactions executed concurrently produces the same result as if the transactions executed sequentially one after the other. If `SERIALIZABLE` is specified, Greenplum Database falls back to `REPEATABLE READ`. The MVCC Snapshot Isolation (SI) model prevents dirty reads, non-repeatable reads, and phantom reads without expensive locking, but there are other interactions that can occur between some `SERIALIZABLE` transactions in Greenplum Database that prevent them from being truly serializable. These anomalies can often be attributed to the fact that Greenplum Database does not perform *predicate locking*, which means that a write in one transaction can affect the result of a previous read in another concurrent transaction.

> **Note:** The PostgreSQL 9.1 `SERIALIZABLE` isolation level introduces a new Serializable Snapshot Isolation (SSI) model, which is fully compliant with the SQL standard definition of serializable transactions. This model is not available in Greenplum Database. SSI monitors concurrent transactions for conditions that could cause serialization anomalies. When potential serialization problems are found, one transaction is allowed to commit and others are rolled back and must be retried.

Greenplum Database transactions that run concurrently should be examined to identify interactions that may update the same data concurrently. Problems identified can be prevented by using explicit table locks or by requiring the conflicting transactions to update a dummy row introduced to represent the conflict.

The SQL `SET TRANSACTION ISOLATION LEVEL` statement sets the isolation mode for the current transaction. The mode must be set before any `SELECT`, `INSERT`, `DELETE`, `UPDATE`, or `COPY` statements:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
...
COMMIT;
```

The isolation mode can also be specified as part of the `BEGIN` statement:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

The default transaction isolation mode can be changed for a session by setting the *default_transaction_isolation* configuration property.

Removing Dead Rows from Tables

Updating or deleting a row leaves an expired version of the row in the table. When an expired row is no longer referenced by any active transactions, it can be removed and the space it occupied can be reused. The `VACUUM` command marks the space used by expired rows for reuse.

When expired rows accumulate in a table, the disk files must be extended to accommodate new rows. Performance suffers due to the increased disk I/O required to execute queries. This condition is called *bloat* and it should be managed by regularly vacuuming tables.

The `VACUUM` command (without `FULL`) can run concurrently with other queries. It marks the space previously used by the expired rows as free, and updates the free space map. When Greenplum Database later needs space for new rows, it first consults the table's free space map to find pages with available space. If none are found, new pages will be appended to the file.

`VACUUM` (without `FULL`) does not consolidate pages or reduce the size of the table on disk. The space it recovers is only available through the free space map. To prevent disk files from growing, it is important to run `VACUUM` often enough. The frequency of required `VACUUM` runs depends on the frequency of updates and deletes in the table (inserts only ever add new rows). Heavily updated tables might require several `VACUUM` runs per day, to ensure that the available free space can be found through the free space map. It is also important to run `VACUUM` after running a transaction that updates or deletes a large number of rows.

The `VACUUM FULL` command rewrites the table without expired rows, reducing the table to its minimum size. Every page in the table is checked, and visible rows are moved up into pages which are not yet fully packed. Empty pages are discarded. The table is locked until `VACUUM FULL` completes. This is very expensive compared to the regular `VACUUM` command, and can be avoided or postponed by vacuuming regularly. It is best to run `VACUUM FULL` during a maintenance period. An alternative to `VACUUM FULL` is to recreate the table with a `CREATE TABLE AS` statement and then drop the old table.

You can run `VACUUM VERBOSE` *`tablename`* to get a report, by segment, of the number of dead rows removed, the number of pages affected, and the number of pages with usable free space.

Query the `pg_class` system table to find out how many pages a table is using across all segments. Be sure to `ANALYZE` the table first to get accurate data.

```
SELECT relname, relpages, reltuples FROM pg_class WHERE relname='tablename';
```

Another useful tool is the `gp_bloat_diag` view in the `gp_toolkit` schema, which identifies bloat in tables by comparing the actual number of pages used by a table to the expected number. See "The gp_toolkit Administrative Schema" in the *Greenplum Database Reference Guide* for more about `gp_bloat_diag`.

Example of Managing Transaction IDs

For Greenplum Database, the transaction ID (XID) value an incrementing 32-bit (2^{32}) value. The maximum unsigned 32-bit value is 4,294,967,295, or about four billion. The XID values restart at 3 after the maximum is reached. Greenplum Database handles the limit of XID values with two features:

- Calculations on XID values using modulo-2^{32} arithmetic that allow Greenplum Database to reuse XID values. The modulo calculations determine the order of transactions, whether one transaction has occurred before or after another, based on the XID.

  Every XID value can have up to two billion (2^{31}) XID values that are considered previous transactions and two billion (2^{31} -1) XID values that are considered newer transactions. The XID values can be considered a circular set of values with no endpoint similar to a 24 hour clock.

  Using the Greenplum Database modulo calculations, as long as two XIDs are within 2^{31} transactions of each other, comparing them yields the correct result.
- A frozen XID value that Greenplum Database uses as the XID for current (visible) data rows. Setting a row's XID to the frozen XID performs two functions.

- When Greenplum Database compares XIDs using the modulo calculations, the frozen XID is always smaller, earlier, when compared to any other XID. If a row's XID is not set to the frozen XID and 2^{31} new transactions are executed, the row appears to be executed in the future based on the modulo calculation.
- When the row's XID is set to the frozen XID, the original XID can be used, without duplicating the XID. This keeps the number of data rows on disk with assigned XIDs below (2^{32}).

  Note: Greenplum Database assigns XID values only to transactions that involve DDL or DML operations, which are typically the only transactions that require an XID.

Simple MVCC Example

This is a simple example of the concepts of a MVCC database and how it manages data and transactions with transaction IDs. This simple MVCC database example consists of a single table:

- The table is a simple table with 2 columns and 4 rows of data.
- The valid transaction ID (XID) values are from 0 up to 9, after 9 the XID restarts at 0.
- The frozen XID is -2. This is different than the Greenplum Database frozen XID.
- Transactions are performed on a single row.
- Only insert and update operations are performed.
- All updated rows remain on disk, no operations are performed to remove obsolete rows.

The example only updates the amount values. No other changes to the table.

The example shows these concepts.

- *How transaction IDs are used to manage multiple, simultaneous transactions on a table.*
- *How transaction IDs are managed with the frozen XID*
- *How the modulo calculation determines the order of transactions based on transaction IDs*

Managing Simultaneous Transactions

This table is the initial table data on disk with no updates. The table contains two database columns for transaction IDs, `xmin` (transaction that created the row) and `xmax` (transaction that updated the row). In the table, changes are added, in order, to the bottom of the table.

Table 12: Example Table

item	amount	xmin	xmax
widget	100	0	null
giblet	200	1	null
sprocket	300	2	null
gizmo	400	3	null

The next table shows the table data on disk after some updates on the amount values have been performed.

xid = 4: `update tbl set amount=208 where item = 'widget'`
xid = 5: `update tbl set amount=133 where item = 'sprocket'`
xid = 6: `update tbl set amount=16 where item = 'widget'`

In the next table, the bold items are the current rows for the table. The other rows are obsolete rows, table data that on disk but is no longer current. Using the xmax value, you can determine the current rows of the table by selecting the rows with `null` value. Greenplum Database uses a slightly different method to determine current table rows.

Table 13: Example Table with Updates

item	amount	xmin	xmax
widget	100	0	4
giblet	**200**	1	**null**
sprocket	300	2	5
gizmo	**400**	3	**null**
widget	208	4	6
sproket	**133**	5	**null**
widget	**16**	6	**null**

The simple MVCC database works with XID values to determine the state of the table. For example, both these independent transactions execute concurrently.

- `UPDATE` command changes the sprocket amount value to `133` (xmin value `5`)
- `SELECT` command returns the value of sprocket.

During the `UPDATE` transaction, the database returns the value of sprocket `300`, until the `UPDATE` transaction completes.

Managing XIDs and the Frozen XID

For this simple example, the database is close to running out of available XID values. When Greenplum Database is close to running out of available XID values, Greenplum Database takes these actions.

- Greenplum Database issues a warning stating that the database is running out of XID values.

```
WARNING: database "database_name" must be vacuumed
 within number_of_transactions transactions
```

- Before the last XID is assigned, Greenplum Database stops accepting transactions to prevent assigning an XID value twice and issues this message.

```
FATAL: database is not accepting commands to avoid wraparound data loss in
 database "database_name"
```

To manage transaction IDs and table data that is stored on disk, Greenplum Database provides the `VACUUM` command.

- A `VACUUM` operation frees up XID values so that a table can have more than 10 rows by changing the xmin values to the frozen XID.
- A `VACUUM` operation manages obsolete or deleted table rows on disk. This database's `VACUUM` command changes the XID values `obsolete` to indicate obsolete rows. A Greenplum Database `VACUUM` operation, without the `FULL` option, deletes the data opportunistically to remove rows on disk with minimal impact to performance and data availability.

For the example table, a `VACUUM` operation has been performed on the table. The command updated table data on disk. This version of the `VACUUM` command performs slightly differently than the Greenplum Database command, but the concepts are the same.

- For the widget and sprocket rows on disk that are no longer current, the rows have been marked as `obsolete`.
- For the giblet and gizmo rows that are current, the xmin has been changed to the frozen XID.

The values are still current table values (the row's xmax value is `null`). However, the table row is visible to all transactions because the xmin value is frozen XID value that is older than all other XID values when modulo calculations are performed.

After the `VACUUM` operation, the XID values `0`, `1`, `2`, and `3` available for use.

Table 14: Example Table after VACUUM

item	amount	xmin	xmax
widget	100	obsolete	obsolete
giblet	**200**	-2	**null**
sprocket	300	obsolete	obsolete
gizmo	**400**	-2	**null**
widget	208	4	6
sproket	**133**	5	**null**
widget	**16**	6	**null**

When a row disk with the xmin value of `-2` is updated, the xmax value is replaced with the transaction XID as usual, and the row on disk is considered obsolete after any concurrent transactions that access the row have completed.

Obsolete rows can be deleted from disk. For Greenplum Database, the `VACUUM` command, with `FULL` option, does more extensive processing to reclaim disk space.

Example of XID Modulo Calculations

The next table shows the table data on disk after more `UPDATE` transactions. The XID values have rolled over and start over at `0`. No additional `VACUUM` operations have been performed.

Table 15: Example Table with Wrapping XID

item	amount	xmin	xmax
widget	100	obsolete	obsolete
giblet	200	-2	1
sprocket	300	obsolete	obsolete
gizmo	400	-2	9
widget	208	4	6
sproket	**133**	5	**null**
widget	16	6	7
widget	**222**	7	**null**
giblet	233	8	0
gizmo	**18**	9	**null**
giblet	88	0	1
giblet	**44**	1	**null**

When performing the modulo calculations that compare XIDs, Greenplum Database, considers the XIDs of the rows and the current range of available XIDs to determine if XID wrapping has occurred between row XIDs.

For the example table XID wrapping has occurred. The XID `1` for giblet row is a later transaction than the XID `7` for widget row based on the modulo calculations for XID values even though the XID value `7` is larger than `1`.

For the widget and sprocket rows, XID wrapping has not occurred and XID `7` is a later transaction than XID `5`.

About Parallel Data Loading

This topic provides a short introduction to Greenplum Database data loading features.

In a large scale, multi-terabyte data warehouse, large amounts of data must be loaded within a relatively small maintenance window. Greenplum supports fast, parallel data loading with its external tables feature. Administrators can also load external tables in single row error isolation mode to filter bad rows into a separate error log while continuing to load properly formatted rows. Administrators can specify an error threshold for a load operation to control how many improperly formatted rows cause Greenplum to abort the load operation.

By using external tables in conjunction with Greenplum Database's parallel file server (`gpfdist`), administrators can achieve maximum parallelism and load bandwidth from their Greenplum Database system.

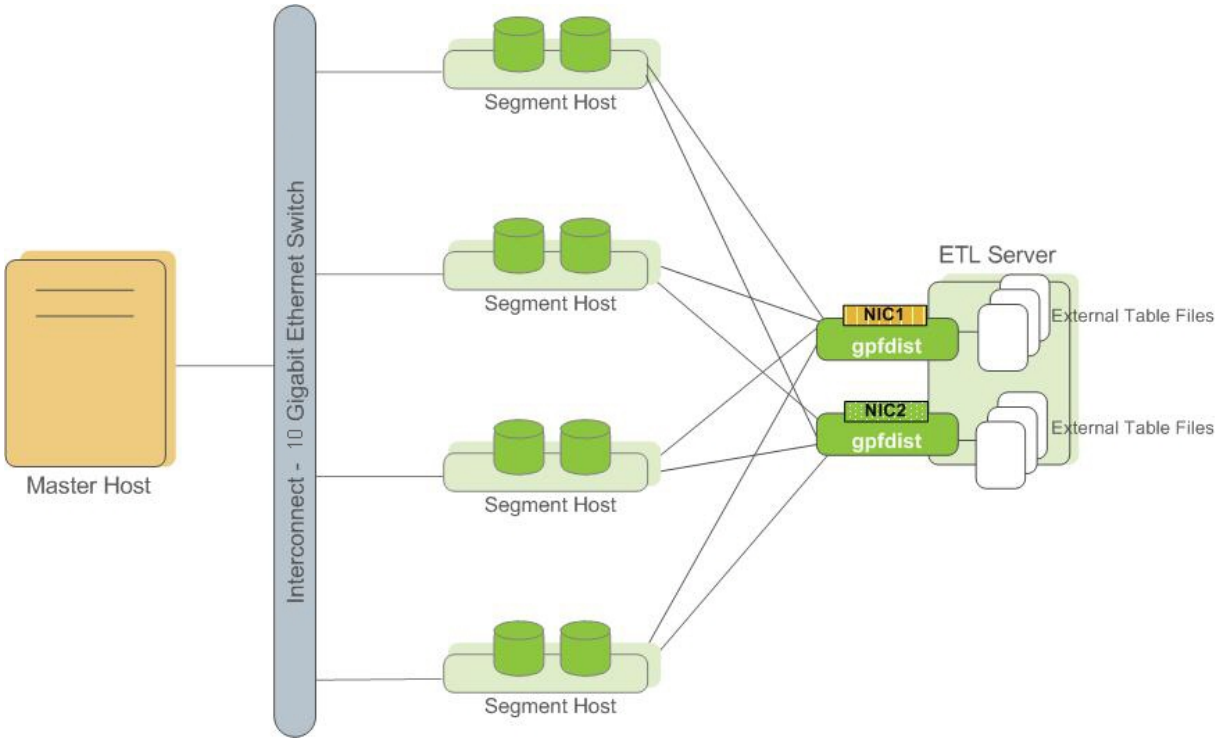


Figure 12: External Tables Using Greenplum Parallel File Server (gpfdist)

Another Greenplum utility, `gpload`, runs a load task that you specify in a YAML-formatted control file. You describe the source data locations, format, transformations required, participating hosts, database destinations, and other particulars in the control file and `gpload` executes the load. This allows you to describe a complex task and execute it in a controlled, repeatable fashion.

About Redundancy and Failover in Greenplum Database

This topic provides a high-level overview of Greenplum Database high availability features.

You can deploy Greenplum Database without a single point of failure by mirroring components. The following sections describe the strategies for mirroring the main components of a Greenplum system. For a more detailed overview of Greenplum high availability features, see *Overview of Greenplum Database High Availability*.

> **Important:** When data loss is not acceptable for a Greenplum Database cluster, Greenplum master and segment mirroring is recommended. If mirroring is not enabled then Greenplum stores only one copy of the data, so the underlying storage media provides the only guarantee for data availability and correctness in the event of a hardware failure.
>
> Kubernetes enables quick recovery from both pod and host failures, and Kubernetes storage services provide a high level of availability for the underlying data. Furthermore, virtualized environments make it difficult to ensure the anti-affinity guarantees required for Greenplum mirroring solutions. For these reasons, mirrorless deployments are fully supported with Greenplum for Kubernetes. Other deployment environments are generally not supported for production use unless both Greenplum master and segment mirroring are enabled.

About Segment Mirroring

When you deploy your Greenplum Database system, you can configure *mirror* segment instances. Mirror segments allow database queries to fail over to a backup segment if the primary segment becomes unavailable. The mirror segment is kept current by a transaction log replication process, which synchronizes the data between the primary and mirror instances. Mirroring is strongly recommended for production systems and required for Pivotal support.

As a best practice, the secondary (mirror) segment instance must always reside on a different host than its primary segment instance to protect against a single host failure. In virtualized environments, the secondary (mirror) segment must always reside on a different storage system than the primary. Mirror segments can be arranged over the remaining hosts in the cluster in configurations designed to maximize availability, or minimize the performance degradation when hosts or multiple primary segments fail.

Two standard mirroring configurations are available when you initialize or expand a Greenplum system. The default configuration, called *group mirroring*, places all the mirrors for a host's primary segments on one other host in the cluster. The other standard configuration, *spread mirroring*, can be selected with a command-line option. Spread mirroring spreads each host's mirrors over the remaining hosts and requires that there are more hosts in the cluster than primary segments per host.

Figure 13: Spread Mirroring in Greenplum Database shows how table data is distributed across segments when spread mirroring is configured.

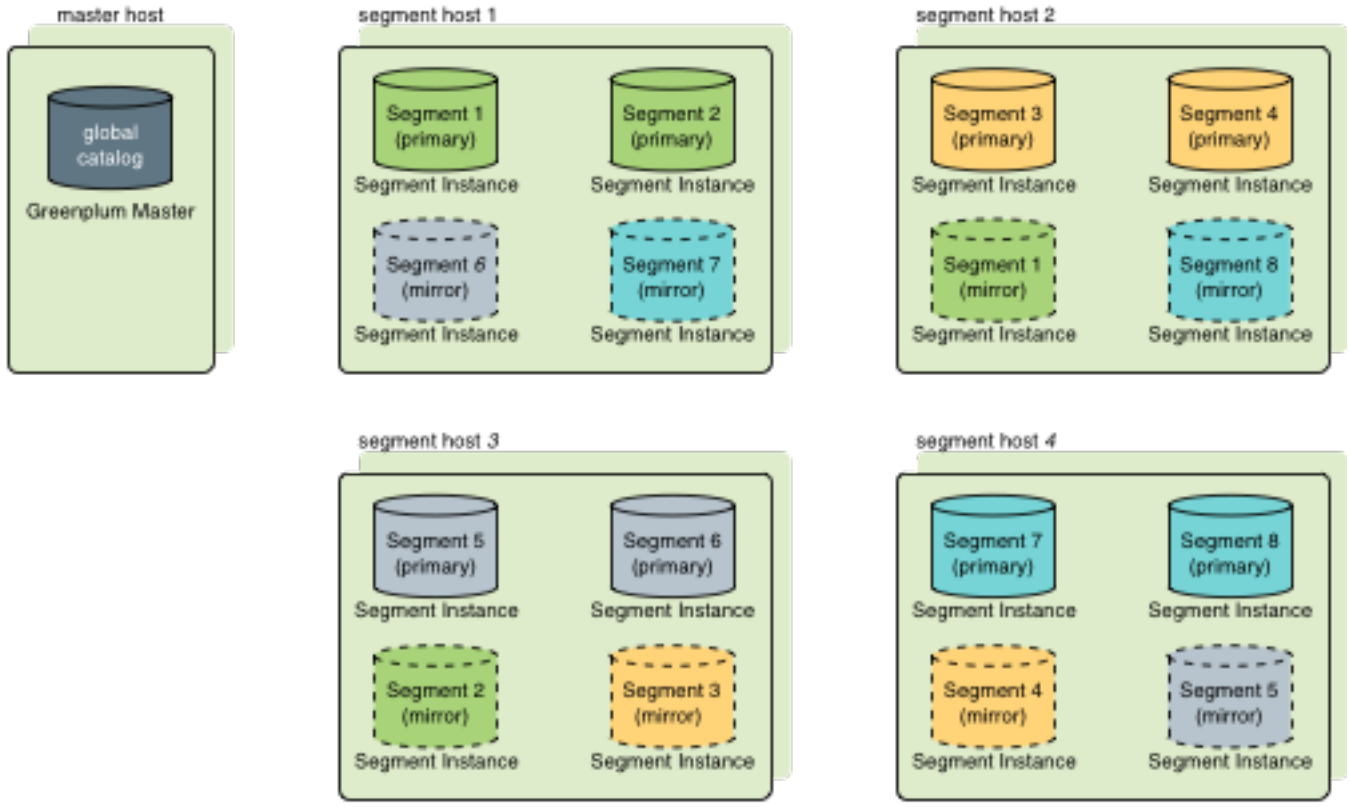


Figure 13: Spread Mirroring in Greenplum Database

Segment Failover and Recovery

When segment mirroring is enabled in a Greenplum Database system, the system will automatically fail over to the *mirror segment* instance if a *primary segment* instance becomes unavailable. A Greenplum Database system can remain operational if a segment instance or host goes down as long as all the data is available on the remaining active segment instances.

If the master cannot connect to a segment instance, it marks that segment instance as down in the Greenplum Database system catalog and brings up the mirror segment in its place. A failed segment instance will remain out of operation until an administrator takes steps to bring that segment back online. An administrator can recover a failed segment while the system is up and running. The recovery process copies over only the changes that were missed while the segment was out of operation.

If you do not have mirroring enabled, the system will automatically shut down if a segment instance becomes invalid. You must recover all failed segments before operations can continue.

About Master Mirroring

You can also optionally deploy a backup or mirror of the master instance on a separate host from the master host. The backup master instance (the *standby master*) serves as a *warm standby* in the event that the primary master host becomes non-operational. The standby master is kept current by a transaction log replication process, which synchronizes the data between the primary and standby master.

If the primary master fails, the log replication process stops, and the standby master can be activated in its place. The switchover does not happen automatically, but must be triggered externally. Upon activation of the standby master, the replicated logs are used to reconstruct the state of the master host at the time of the last successfully committed transaction. The activated standby master effectively becomes the Greenplum Database master, accepting client connections on the master port (which must be set to the same port number on the master host and the backup master host).

Since the master does not contain any user data, only the system catalog tables need to be synchronized between the primary and backup copies. When these tables are updated, changes are automatically copied over to the standby master to ensure synchronization with the primary master.

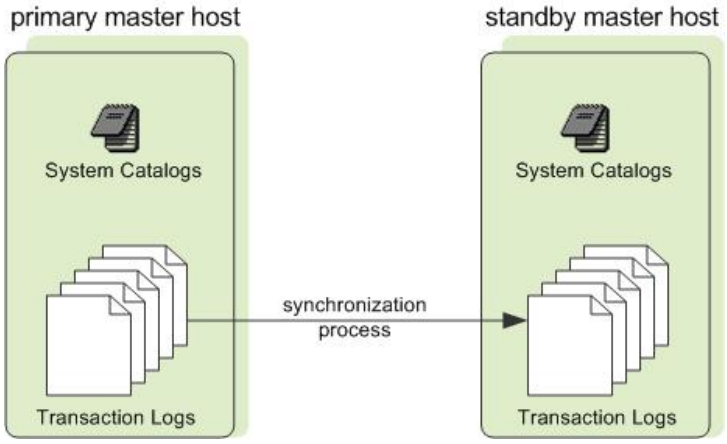


Figure 14: Master Mirroring in Greenplum Database

About Interconnect Redundancy

The *interconnect* refers to the inter-process communication between the segments and the network infrastructure on which this communication relies. You can achieve a highly available interconnect using by deploying dual Gigabit Ethernet switches on your network and redundant Gigabit connections to the Greenplum Database host (master and segment) servers. For performance reasons, 10-Gb Ethernet, or faster, is recommended.

About Database Statistics in Greenplum Database

An overview of statistics gathered by the *ANALYZE* command in Greenplum Database.

Statistics are metadata that describe the data stored in the database. The query optimizer needs up-to-date statistics to choose the best execution plan for a query. For example, if a query joins two tables and one of them must be broadcast to all segments, the optimizer can choose the smaller of the two tables to minimize network traffic.

The statistics used by the optimizer are calculated and saved in the system catalog by the `ANALYZE` command. There are three ways to initiate an analyze operation:

- You can run the `ANALYZE` command directly.
- You can run the `analyzedb` management utility outside of the database, at the command line.
- An automatic analyze operation can be triggered when DML operations are performed on tables that have no statistics or when a DML operation modifies a number of rows greater than a specified threshold.

These methods are described in the following sections. The `VACUUM ANALYZE` command is another way to initiate an analyze operation, but its use is discouraged because vacuum and analyze are different operations with different purposes.

Calculating statistics consumes time and resources, so Greenplum Database produces estimates by calculating statistics on samples of large tables. In most cases, the default settings provide the information needed to generate correct execution plans for queries. If the statistics produced are not producing optimal query execution plans, the administrator can tune configuration parameters to produce more accurate stastistics by increasing the sample size or the granularity of statistics saved in the system catalog. Producing more accurate statistics has CPU and storage costs and may not produce better plans, so it is important to view explain plans and test query performance to ensure that the additional statistics-related costs result in better query performance.

System Statistics

Table Size

The query planner seeks to minimize the disk I/O and network traffic required to execute a query, using estimates of the number of rows that must be processed and the number of disk pages the query must access. The data from which these estimates are derived are the `pg_class` system table columns `reltuples` and `relpages`, which contain the number of rows and pages at the time a `VACUUM` or `ANALYZE` command was last run. As rows are added or deleted, the numbers become less accurate. However, an accurate count of disk pages is always available from the operating system, so as long as the ratio of `reltuples` to `relpages` does not change significantly, the optimizer can produce an estimate of the number of rows that is sufficiently accurate to choose the correct query execution plan.

When the `reltuples` column differs significantly from the row count returned by `SELECT COUNT(*)`, an analyze should be performed to update the statistics.

When a `REINDEX` command finishes recreating an index, the `relpages` and `reltuples` columns are set to zero. The `ANALYZE` command should be run on the base table to update these columns.

The pg_statistic System Table and pg_stats View

The `pg_statistic` system table holds the results of the last `ANALYZE` operation on each database table. There is a row for each column of every table. It has the following columns:

`starelid`

The object ID of the table or index the column belongs to.

`staattnum`

The number of the described column, beginning with 1.

`stainherit`

If true, the statistics include inheritance child columns, not just the values in the specified relation.

`stanullfrac`

The fraction of the column's entries that are null.

`stawidth`

The average stored width, in bytes, of non-null entries.

`stadistinct`

A positive number is an estimate of the number of distinct values in the column; the number is not expected to vary with the number of rows. A negative value is the number of distinct values divided by the number of rows, that is, the ratio of rows with distinct values for the column, negated. This form is used when the number of distinct values increases with the number of rows. A unique column, for example, has an `n_distinct` value of -1.0. Columns with an average width greater than 1024 are considered unique.

`stakind`*N*

A code number indicating the kind of statistics stored in the *N*th slot of the `pg_statistic` row.

`staop`*N*

An operator used to derive the statistics stored in the *N*th slot. For example, a histogram slot would show the < operator that defines the sort order of the data.

`stanumbers`*N*

float4 array containing numerical statistics of the appropriate kind for the *N*th slot, or `NULL` if the slot kind does not involve numerical values.

`stavalues`*`N`*

Column data values of the appropriate kind for the *N*th slot, or `NULL` if the slot kind does not store any data values. Each array's element values are actually of the specific column's data type, so there is no way to define these columns' types more specifically than *anyarray*.

The statistics collected for a column vary for different data types, so the `pg_statistic` table stores statistics that are appropriate for the data type in four *slots*, consisting of four columns per slot. For example, the first slot, which normally contains the most common values for a column, consists of the columns `stakind1`, `staop1`, `stanumbers1`, and `stavalues1`.

The `stakind`*`N`* columns each contain a numeric code to describe the type of statistics stored in their slot. The `stakind` code numbers from 1 to 99 are reserved for core PostgreSQL data types. Greenplum Database uses code numbers 1, 2, 3, 4, 5, and 99. A value of 0 means the slot is unused. The following table describes the kinds of statistics stored for the three codes.

Table 16: Contents of pg_statistic "slots"

stakind Code	Description
1	*Most CommonValues (MCV) Slot* • `staop` contains the object ID of the "=" operator, used to decide whether values are the same or not. • `stavalues` contains an array of the *K* most common non-null values appearing in the column. • `stanumbers` contains the frequencies (fractions of total row count) of the values in the `stavalues` array. The values are ordered in decreasing frequency. Since the arrays are variable-size, *K* can be chosen by the statistics collector. Values must occur more than once to be added to the `stavalues` array; a unique column has no MCV slot.
2	*Histogram Slot* – describes the distribution of scalar data. • `staop` is the object ID of the "<" operator, which describes the sort ordering. • `stavalues` contains *M* (where *M*>=2) non-null values that divide the non-null column data values into *M*-1 bins of approximately equal population. The first `stavalues` item is the minimum value and the last is the maximum value. • `stanumbers` is not used and should be `NULL`. If a Most Common Values slot is also provided, then the histogram describes the data distribution after removing the values listed in the MCV array. (It is a *compressed histogram* in the technical parlance). This allows a more accurate representation of the distribution of a column with some very common values. In a column with only a few distinct values, it is possible that the MCV list describes the entire data population; in this case the histogram reduces to empty and should be omitted.
3	*Correlation Slot* – describes the correlation between the physical order of table tuples and the ordering of data values of this column. • `staop` is the object ID of the "<" operator. As with the histogram, more than one entry could theoretically appear. • `stavalues` is not used and should be `NULL`. • `stanumbers` contains a single entry, the correlation coefficient between the sequence of data values and the sequence of their actual tuple positions. The coefficient ranges from +1 to -1.

stakind Code	Description
4	*Most Common Elements Slot* - is similar to a Most Common Values (MCV) Slot, except that it stores the most common non-null *elements* of the column values. This is useful when the column datatype is an array or some other type with identifiable elements (for instance, `tsvector`). • `staop` contains the equality operator appropriate to the element type. • `stavalues` contains the most common element values. • `stanumbers` contains common element frequencies. Frequencies are measured as the fraction of non-null rows the element value appears in, not the frequency of all rows. Also, the values are sorted into the element type's default order (to support binary search for a particular value). Since this puts the minimum and maximum frequencies at unpredictable spots in `stanumbers`, there are two extra members of `stanumbers` that hold copies of the minimum and maximum frequencies. Optionally, there can be a third extra member that holds the frequency of null elements (the frequency is expressed in the same terms: the fraction of non-null rows that contain at least one null element). If this member is omitted, the column is presumed to contain no `NULL` elements. **Note:** For `tsvector` columns, the `stavalues` elements are of type `text`, even though their representation within `tsvector` is not exactly `text`.
5	*Distinct Elements Count Histogram Slot* - describes the distribution of the number of distinct element values present in each row of an array-type column. Only non-null rows are considered, and only non-null elements. • `staop` contains the equality operator appropriate to the element type. • `stavalues` is not used and should be `NULL`. • `stanumbers` contains information about distinct elements. The last member of `stanumbers` is the average count of distinct element values over all non-null rows. The preceding *M* (where *M* >=2) members form a histogram that divides the population of distinct-elements counts into *M*-1 bins of approximately equal population. The first of these is the minimum observed count, and the last the maximum.
99	*Hyperloglog Slot* - for child leaf partitions of a partitioned table, stores the `hyperloglog_counter` created for the sampled data. The `hyperloglog_counter` data structure is converted into a `bytea` and stored in a `stavalues5` slot of the `pg_statistic` catalog table.

The `pg_stats` view presents the contents of `pg_statistic` in a friendlier format. The `pg_stats` view has the following columns:

schemaname	The name of the schema containing the table.
tablename	The name of the table.
attname	The name of the column this row describes.
inherited	If true, the statistics include inheritance child columns.
null_frac	The fraction of column entries that are null.
avg_width	The average storage width in bytes of the column's entries, calculated as `avg(pg_column_size(`*`column_name`*`))`.

n_distinct	A positive number is an estimate of the number of distinct values in the column; the number is not expected to vary with the number of rows. A negative value is the number of distinct values divided by the number of rows, that is, the ratio of rows with distinct values for the column, negated. This form is used when the number of distinct values increases with the number of rows. A unique column, for example, has an `n_distinct` value of -1.0. Columns with an average width greater than 1024 are considered unique.
most_common_vals	An array containing the most common values in the column, or null if no values seem to be more common. If the `n_distinct` column is -1, `most_common_vals` is null. The length of the array is the lesser of the number of actual distinct column values or the value of the `default_statistics_target` configuration parameter. The number of values can be overridden for a column using `ALTER TABLE` *`table`* `SET COLUMN` *`column`* `SET STATISTICS` *`N`*.
most_common_freqs	An array containing the frequencies of the values in the `most_common_vals` array. This is the number of occurrences of the value divided by the total number of rows. The array is the same length as the `most_common_vals` array. It is null if `most_common_vals` is null.
histogram_bounds	An array of values that divide the column values into groups of approximately the same size. A histogram can be defined only if there is a `max()` aggregate function for the column. The number of groups in the histogram is the same as the `most_common_vals` array size.
correlation	Greenplum Database does not calculate the correlation statistic.
most_common_elems	An array that contains the most common element values.
most_common_elem_freqs	An array that contains common element frequencies.
elem_count_histogram	An array that describes the distribution of the number of distinct element values present in each row of an array-type column.

Newly created tables and indexes have no statistics. You can check for tables with missing statistics using the `gp_stats_missing` view, which is in the `gp_toolkit` schema:

```
SELECT * from gp_toolkit.gp_stats_missing;
```

Sampling

When calculating statistics for large tables, Greenplum Database creates a smaller table by sampling the base table. If the table is partitioned, samples are taken from all partitions.

Updating Statistics

Running ANALYZE with no arguments updates statistics for all tables in the database. This could take a very long time, so it is better to analyze tables selectively after data has changed. You can also analyze a subset of the columns in a table, for example columns used in joins, WHERE clauses, SORT clauses, GROUP BY clauses, or HAVING clauses.

Analyzing a severely bloated table can generate poor statistics if the sample contains empty pages, so it is good practice to vacuum a bloated table before analyzing it.

See the *SQL Command Reference* in the *Greenplum Database Reference Guide* for details of running the ANALYZE command.

Refer to the *Greenplum Database Management Utility Reference* for details of running the `analyzedb` command.

Analyzing Partitioned Tables

When the ANALYZE command is run on a partitioned table, it analyzes each child leaf partition table, one at a time. You can run ANALYZE on just new or changed partition files to avoid analyzing partitions that have not changed.

The `analyzedb` command-line utility skips unchanged partitions automatically. It also runs concurrent sessions so it can analyze several partitions concurrently. It runs five sessions by default, but the number of sessions can be set from 1 to 10 with the `-p` command-line option. Each time `analyzedb` runs, it saves state information for append-optimized tables and partitions in the `db_analyze` directory in the master data directory. The next time it runs, `analyzedb` compares the current state of each table with the saved state and skips analyzing a table or partition if it is unchanged. Heap tables are always analyzed.

If GPORCA is enabled (the default), you also need to run ANALYZE or ANALYZE ROOTPARTITION to refresh the root partition statistics. GPORCA requires statistics at the root level for partitioned tables. The Postgres Planner does not use these statistics.

The time to analyze a partitioned table is similar to the time to analyze a non-partitioned table with the same data since ANALYZE ROOTPARTITION does not collect statistics on the leaf partitions (the data is only sampled).

The Greenplum Database server configuration parameter *optimizer_analyze_root_partition* affects when statistics are collected on the root partition of a partitioned table. If the parameter is on (the default), the ROOTPARTITION keyword is not required to collect statistics on the root partition when you run ANALYZE. Root partition statistics are collected when you run ANALYZE on the root partition, or when you run ANALYZE on a child leaf partition of the partitioned table and the other child leaf partitions have statistics. If the parameter is off, you must run ANALYZE ROOTPARTITION to collect root partition statistics.

If you do not intend to execute queries on partitioned tables with GPORCA (setting the server configuration parameter *optimizer* to off), you can also set the server configuration parameter optimizer_analyze_root_partition to off to limit when ANALYZE updates the root partition statistics.

Configuring Statistics

There are several options for configuring Greenplum Database statistics collection.

Statistics Target

The statistics target is the size of the most_common_vals, most_common_freqs, and histogram_bounds arrays for an individual column. By default, the target is 25. The default target can be changed by setting a server configuration parameter and the target can be set for any column using the ALTER TABLE command. Larger values increase the time needed to do ANALYZE, but may improve the quality of the Postgres Planner estimates.

Set the system default statistics target to a different value by setting the `default_statistics_target` server configuration parameter. The default value is usually sufficient, and you should only raise or lower it if your tests demonstrate that query plans improve with the new target. For example, to raise the default statistics target from 100 to 150 you can use the `gpconfig` utility:

```
gpconfig -c default_statistics_target -v 150
```

The statististics target for individual columns can be set with the `ALTER TABLE` command. For example, some queries can be improved by increasing the target for certain columns, especially columns that have irregular distributions. You can set the target to zero for columns that never contribute to query otpimization. When the target is 0, `ANALYZE` ignores the column. For example, the following `ALTER TABLE` command sets the statistics target for the `notes` column in the `emp` table to zero:

```
ALTER TABLE emp ALTER COLUMN notes SET STATISTICS 0;
```

The statistics target can be set in the range 0 to 1000, or set it to -1 to revert to using the system default statistics target.

Setting the statistics target on a parent partition table affects the child partitions. If you set statistics to 0 on some columns on the parent table, the statistics for the same columns are set to 0 for all children partitions. However, if you later add or exchange another child partition, the new child partition will use either the default statistics target or, in the case of an exchange, the previous statistics target. Therefore, if you add or exchange child partitions, you should set the statistics targets on the new child table.

Automatic Statistics Collection

Greenplum Database can be set to automatically run `ANALYZE` on a table that either has no statistics or has changed significantly when certain operations are performed on the table. For partitioned tables, automatic statistics collection is only triggered when the operation is run directly on a leaf table, and then only the leaf table is analyzed.

Automatic statistics collection has three modes:

- `none` disables automatic statistics collection.
- `on_no_stats` triggers an analyze operation for a table with no existing statistics when any of the commands `CREATE TABLE AS SELECT`, `INSERT`, or `COPY` are executed on the table.
- `on_change` triggers an analyze operation when any of the commands `CREATE TABLE AS SELECT`, `UPDATE`, `DELETE`, `INSERT`, or `COPY` are executed on the table and the number of rows affected exceeds the threshold defined by the `gp_autostats_on_change_threshold` configuration parameter.

The automatic statistics collection mode is set separately for commands that occur within a procedural language function and commands that execute outside of a function:

- The `gp_autostats_mode` configuration parameter controls automatic statistics collection behavior outside of functions and is set to `on_no_stats` by default.
- The `gp_autostats_mode_in_functions` parameter controls the behavior when table operations are performed within a procedural language function and is set to `none` by default.

With the `on_change` mode, `ANALYZE` is triggered only if the number of rows affected exceeds the threshold defined by the `gp_autostats_on_change_threshold` configuration parameter. The default value for this parameter is a very high value, 2147483647, which effectively disables automatic statistics collection; you must set the threshold to a lower number to enable it. The `on_change` mode could trigger large, unexpected analyze operations that could disrupt the system, so it is not recommended to set it globally. It could be useful in a session, for example to automatically analyze a table following a load.

To disable automatic statistics collection outside of functions, set the `gp_autostats_mode` parameter to `none`:

```
gpconfigure -c gp_autostats_mode -v none
```

To enable automatic statistics collection in functions for tables that have no statistics, change `gp_autostats_mode_in_functions` to `on_no_stats`:

```
gpconfigure -c gp_autostats_mode_in_functions -v on_no_stats
```

Set the `log_autostats` system configuration parameter to on if you want to log automatic statistics collection operations.

Managing a Greenplum System

This section describes basic system administration tasks performed by a Greenplum Database system administrator.

This section contains the following topics:

About the Greenplum Database Release Version Number

Greenplum Database version numbers and they way they change identify what has been modified from one Greenplum Database release to the next.

A Greenplum Database release version number takes the format *x.y.z*, where:

- *x* identifies the Major version number
- *y* identifies the Minor version number
- *z* identifies the Patch version number

Greenplum Database releases that have the same Major release number are guaranteed to be backwards compatible. Greenplum Database increments the Major release number when the catalog changes or when incompatible feature changes or new features are introduced. Previously deprecated functionality may be removed in a major release.

The Minor release number for a given Major release increments when backwards compatible new features are introduced or when a Greenplum Database feature is deprecated. (Previously deprecated functionality will never be removed in a minor release.)

Greenplum Database increments the Patch release number for a given Minor release for backwards-compatible bug fixes.

Starting and Stopping Greenplum Database

In a Greenplum Database DBMS, the database server instances (the master and all segments) are started or stopped across all of the hosts in the system in such a way that they can work together as a unified DBMS.

Because a Greenplum Database system is distributed across many machines, the process for starting and stopping a Greenplum Database system is different than the process for starting and stopping a regular PostgreSQL DBMS.

Use the `gpstart` and `gpstop` utilities to start and stop Greenplum Database, respectively. These utilities are located in the `$GPHOME/bin` directory on your Greenplum Database master host.

> **Important:** Do not issue a `kill` command to end any Postgres process. Instead, use the database command `pg_cancel_backend()`.

Issuing a `kill -9` or `kill -11` can introduce database corruption and prevent root cause analysis from being performed.

For information about `gpstart` and `gpstop`, see the *Greenplum Database Utility Guide*.

Starting Greenplum Database

Start an initialized Greenplum Database system by running the `gpstart` utility on the master instance.

Use the `gpstart` utility to start a Greenplum Database system that has already been initialized by the `gpinitsystem` utility, but has been stopped by the `gpstop` utility. The `gpstart` utility starts Greenplum Database by starting all the Postgres database instances on the Greenplum Database cluster. `gpstart` orchestrates this process and performs the process in parallel.

- Run `gpstart` on the master host to start Greenplum Database:

```
$ gpstart
```

Restarting Greenplum Database

Stop the Greenplum Database system and then restart it.

The `gpstop` utility with the `-r` option can stop and then restart Greenplum Database after the shutdown completes.

- To restart Greenplum Database, enter the following command on the master host:

```
$ gpstop -r
```

Reloading Configuration File Changes Only

Reload changes to Greenplum Database configuration files without interrupting the system.

The `gpstop` utility can reload changes to the `pg_hba.conf` configuration file and to *runtime* parameters in the master `postgresql.conf` file and `pg_hba.conf` file without service interruption. Active sessions pick up changes when they reconnect to the database. Many server configuration parameters require a full system restart (`gpstop -r`) to activate. For information about server configuration parameters, see the *Greenplum Database Reference Guide*.

- Reload configuration file changes without shutting down the system using the `gpstop` utility:

```
$ gpstop -u
```

Starting the Master in Maintenance Mode

Start only the master to perform maintenance or administrative tasks without affecting data on the segments.

Maintenance mode should only be used with direction from Pivotal Technical Support. For example, you could connect to a database only on the master instance in maintenance mode and edit system catalog settings. For more information about system catalog tables, see the *Greenplum Database Reference Guide*.

1. Run `gpstart` using the -m option:

```
$ gpstart -m
```

2. Connect to the master in maintenance mode to do catalog maintenance. For example:

```
$ PGOPTIONS='-c gp_session_role=utility' psql postgres
```

3. After completing your administrative tasks, stop the master in utility mode. Then, restart it in production mode.

```
$ gpstop -mr
```

Warning:

Incorrect use of maintenance mode connections can result in an inconsistent system state. Only Technical Support should perform this operation.

Stopping Greenplum Database

The `gpstop` utility stops or restarts your Greenplum Database system and always runs on the master host. When activated, `gpstop` stops all `postgres` processes in the system, including the master and all segment instances. The `gpstop` utility uses a default of up to 64 parallel worker threads to bring down the Postgres instances that make up the Greenplum Database cluster. The system waits for any active transactions to finish before shutting down. To stop Greenplum Database immediately, use fast mode.

- To stop Greenplum Database:

```
$ gpstop
```

- To stop Greenplum Database in fast mode:

```
$ gpstop -M fast
```

By default, you are not allowed to shut down Greenplum Database if there are any client connections to the database. Use the `-M fast` option to roll back all in progress transactions and terminate any connections before shutting down.

Stopping Client Processes

Greenplum Database launches a new backend process for each client connection. A Greenplum Database user with SUPERUSER privileges can cancel and terminate these client backend processes.

Canceling a backend process with the `pg_cancel_backend()` function ends a specific queued or active client query. Terminating a backend process with the `pg_terminate_backend()` function terminates a client connection to a database.

The `pg_cancel_backend()` function has two signatures:

- `pg_cancel_backend( pid int4 )`
- `pg_cancel_backend( pid int4, msg text )`

The `pg_terminate_backend()` function has two similar signatures:

- `pg_terminate_backend( pid int4 )`
- `pg_terminate_backend( pid int4, msg text )`

If you provide a `msg`, Greenplum Database includes the text in the cancel message returned to the client. `msg` is limited to 128 bytes; Greenplum Database truncates anything longer.

The `pg_cancel_backend()` and `pg_terminate_backend()` functions return `true` if successful, and `false` otherwise.

To cancel or terminate a backend process, you must first identify the process ID of the backend. You can obtain the process ID from the `pid` column of the `pg_stat_activity` view. For example, to view the process information associated with all running and queued queries:

```
=# SELECT usename, pid, waiting, state, query, datname
     FROM pg_stat_activity;
```

Sample partial query output:

```
 usename |  pid     | waiting | state  |         query          | datname
---------+----------+---------+--------+------------------------+---------
  sammy  |   31861  |    f    | idle   | SELECT * FROM testtbl; | testdb
  billy  |   31905  |    t    | active | SELECT * FROM topten;  | testdb
```

Use the output to identify the process id (`pid`) of the query or client connection.

For example, to cancel the waiting query identified in the sample output above and include `'Admin canceled long-running query.'` as the message returned to the client:

```
=# SELECT pg_cancel_backend(31905 ,'Admin canceled long-running query.');
ERROR:  canceling statement due to user request: "Admin canceled long-
running query."
```

Accessing the Database

This topic describes the various client tools you can use to connect to Greenplum Database, and how to establish a database session.

Establishing a Database Session

Users can connect to Greenplum Database using a PostgreSQL-compatible client program, such as `psql`. Users and administrators *always* connect to Greenplum Database through the *master*; the segments cannot accept client connections.

In order to establish a connection to the Greenplum Database master, you will need to know the following connection information and configure your client program accordingly.

Table 17: Connection Parameters

Connection Parameter	Description	Environment Variable
Application name	The application name that is connecting to the database. The default value, held in the `application_name` connection parameter is *psql*.	`$PGAPPNAME`
Database name	The name of the database to which you want to connect. For a newly initialized system, use the `postgres` database to connect for the first time.	`$PGDATABASE`
Host name	The host name of the Greenplum Database master. The default host is the local host.	`$PGHOST`
Port	The port number that the Greenplum Database master instance is running on. The default is 5432.	`$PGPORT`
User name	The database user (role) name to connect as. This is not necessarily the same as your OS user name. Check with your Greenplum administrator if you are not sure what you database user name is. Note that every Greenplum Database system has one superuser account that is created automatically at initialization time. This account has the same name as the OS name of the user who initialized the Greenplum system (typically `gpadmin`).	`$PGUSER`

Connecting with psql provides example commands for connecting to Greenplum Database.

Supported Client Applications

Users can connect to Greenplum Database using various client applications:

- A number of *Greenplum Database Client Applications* are provided with your Greenplum installation. The `psql` client application provides an interactive command-line interface to Greenplum Database.
- Using standard *Database Application Interfaces*, such as ODBC and JDBC, users can create their own client applications that interface to Greenplum Database.
- Most client tools that use standard database interfaces, such as ODBC and JDBC, can be configured to connect to Greenplum Database.

Greenplum Database Client Applications

Greenplum Database comes installed with a number of client utility applications located in the `$GPHOME/bin` directory of your Greenplum Database master host installation. The following are the most commonly used client utility applications:

Table 18: Commonly used client applications

Name	Usage
`createdb`	create a new database
`createlang`	define a new procedural language
`createuser`	define a new database role
`dropdb`	remove a database
`droplang`	remove a procedural language
`dropuser`	remove a role
`psql`	PostgreSQL interactive terminal
`reindexdb`	reindex a database
`vacuumdb`	garbage-collect and analyze a database

> **Note:** `createlang` and `droplang` are deprecated and might be removed in a future release. These utilities are no longer used in Pivotal Greenplum Database, and they will display an error if you attempt to use them to install a Greenplum procedural language. Use the `CREATE EXTENSION` or `DROP EXTENSION` command instead.

When using these client applications, you must connect to a database through the Greenplum master instance. You will need to know the name of your target database, the host name and port number of the master, and what database user name to connect as. This information can be provided on the command-line using the options `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option, it will be interpreted as the database name first.

All of these options have default values which will be used if the option is not specified. The default host is the local host. The default port number is 5432. The default user name is your OS system user name, as is the default database name. Note that OS user names and Greenplum Database user names are not necessarily the same.

If the default values are not correct, you can set the environment variables `PGDATABASE`, `PGHOST`, `PGPORT`, and `PGUSER` to the appropriate values, or use a `psql ~/.pgpass` file to contain frequently-used passwords.

For information about Greenplum Database environment variables, see the *Greenplum Database Reference Guide*. For information about psql, see the *Greenplum Database Utility Guide*.

Connecting with psql

Depending on the default values used or the environment variables you have set, the following examples show how to access a database via `psql`:

```
$ psql -d gpdatabase -h master_host -p 5432 -U gpadmin
```

```
$ psql gpdatabase
```

```
$ psql
```

If a user-defined database has not yet been created, you can access the system by connecting to the `postgres` database. For example:

```
$ psql postgres
```

After connecting to a database, `psql` provides a prompt with the name of the database to which `psql` is currently connected, followed by the string => (or =# if you are the database superuser). For example:

```
gpdatabase=>
```

At the prompt, you may type in SQL commands. A SQL command must end with a `;` (semicolon) in order to be sent to the server and executed. For example:

```
=> SELECT * FROM mytable;
```

See the *Greenplum Reference Guide* for information about using the `psql` client application and SQL commands and syntax.

Using the PgBouncer Connection Pooler

The PgBouncer utility manages connection pools for PostgreSQL and Greenplum Database connections.

The following topics describe how to set up and use PgBouncer with Greenplum Database. Refer to the *PgBouncer web site* for information about using PgBouncer with PostgreSQL.

- *Overview*
- *Migrating PgBouncer*
- *Configuring PgBouncer*
- *Starting PgBouncer*
- *Managing PgBouncer*

Overview

A database connection pool is a cache of database connections. Once a pool of connections is established, connection pooling eliminates the overhead of creating new database connections, so clients connect much faster and the server load is reduced.

The PgBouncer connection pooler, from the PostgreSQL community, is included in your Greenplum Database installation. PgBouncer is a light-weight connection pool manager for Greenplum and PostgreSQL. PgBouncer maintains a pool for connections for each database and user combination. PgBouncer either creates a new database connection for a client or reuses an existing connection for the same user and database. When the client disconnects, PgBouncer returns the connection to the pool for re-use.

PgBouncer shares connections in one of three pool modes:

- *Session pooling* – When a client connects, a connection is assigned to it as long as it remains connected. When the client disconnects, the connection is placed back into the pool.
- *Transaction pooling* – A connection is assigned to a client for the duration of a transaction. When PgBouncer notices the transaction is done, the connection is placed back into the pool. This mode can be used only with applications that do not use features that depend upon a session.
- *Statement pooling* – Statement pooling is like transaction pooling, but multi-statement transactions are not allowed. This mode is intended to enforce autocommit mode on the client and is targeted for PL/Proxy on PostgreSQL.

You can set a default pool mode for the PgBouncer instance. You can override this mode for individual databases and users.

PgBouncer supports the standard connection interface shared by PostgreSQL and Greenplum Database. The Greenplum Database client application (for example, `psql`) connects to the host and port on which PgBouncer is running rather than the Greenplum Database master host and port.

PgBouncer includes a `psql`-like administration console. Authorized users can connect to a virtual database to monitor and manage PgBouncer. You can manage a PgBouncer daemon process via the admin console. You can also use the console to update and reload PgBouncer configuration at runtime without stopping and restarting the process.

PgBouncer natively supports TLS.

Migrating PgBouncer

When you migrate to a new Greenplum Database version, you must migrate your PgBouncer instance to that in the new Greenplum Database installation.

- **If you are migrating to a Greenplum Database version 5.8.x or earlier**, you can migrate PgBouncer without dropping connections. Launch the new PgBouncer process with the `-R` option and the configuration file that you started the process with:

  ```
  $ pgbouncer -R -d pgbouncer.ini
  ```

  The `-R` (reboot) option causes the new process to connect to the console of the old process through a Unix socket and issue the following commands:

  ```
  SUSPEND;
  SHOW FDS;
  SHUTDOWN;
  ```

  When the new process detects that the old process is gone, it resumes the work with the old connections. This is possible because the `SHOW FDS` command sends actual file descriptors to the new process. If the transition fails for any reason, kill the new process and the old process will resume.
- **If you are migrating to a Greenplum Database version 5.9.0 or later**, you must shut down the PgBouncer instance in your old installation and reconfigure and restart PgBouncer in your new installation.
- If you used stunnel to secure PgBouncer connections in your old installation, you must configure SSL/TLS in your new installation using the built-in TLS capabilities of PgBouncer 1.8.1 and later.
- If you used LDAP authentication in your old installation, you must configure LDAP in your new installation using the built-in PAM integration capabilities of PgBouncer 1.8.1 and later. You must also remove or replace any `ldap://`-prefixed password strings in the `auth_file`.

Configuring PgBouncer

You configure PgBouncer and its access to Greenplum Database via a configuration file. This configuration file, commonly named `pgbouncer.ini`, provides location information for Greenplum databases. The

`pgbouncer.ini` file also specifies process, connection pool, authorized users, and authentication configuration for PgBouncer.

Sample `pgbouncer.ini` file contents:

```
[databases]
postgres = host=127.0.0.1 port=5432 dbname=postgres
pgb_mydb = host=127.0.0.1 port=5432 dbname=mydb

[pgbouncer]
pool_mode = session
listen_port = 6543
listen_addr = 127.0.0.1
auth_type = md5
auth_file = users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = gpadmin
```

Refer to the *pgbouncer.ini* reference page for the PgBouncer configuration file format and the list of configuration properties it supports.

When a client connects to PgBouncer, the connection pooler looks up the the configuration for the requested database (which may be an alias for the actual database) that was specified in the `pgbouncer.ini` configuration file to find the host name, port, and database name for the database connection. The configuration file also identifies the authentication mode in effect for the database.

PgBouncer requires an authentication file, a text file that contains a list of Greenplum Database users and passwords. The contents of the file are dependent on the `auth_type` you configure in the `pgbouncer.ini` file. Passwords may be either clear text or MD5-encoded strings. You can also configure PgBouncer to query the destination database to obtain password information for users that are not in the authentication file.

PgBouncer Authentication File Format

PgBouncer requires its own user authentication file. You specify the name of this file in the `auth_file` property of the `pgbouncer.ini` configuration file. `auth_file` is a text file in the following format:

```
"username1" "password" ...
"username2" "md5abcdef012342345" ...
"username2" "SCRAM-SHA-256$<iterations>:<salt>$<storedkey>:<serverkey>"
```

`auth_file` contains one line per user. Each line must have at least two fields, both of which are enclosed in double quotes (`" "`). The first field identifies the Greenplum Database user name. The second field is either a plain-text password, an MD5-encoded password, or or a SCRAM secret. PgBouncer ignores the remainder of the line.

(The format of `auth_file` is similar to that of the `pg_auth` text file that Greenplum Database uses for authentication information. PgBouncer can work directly with this Greenplum Database authentication file.)

Use an MD5 encoded password. The format of an MD5 encoded password is:

```
"md5" + MD5_encoded(<password><username>)
```

You can also obtain the MD5-encoded passwords of all Greenplum Database users from the `pg_shadow` view.

PostgreSQL SCRAM secret format:

```
SCRAM-SHA-256$<iterations>:<salt>$<storedkey>:<serverkey>
```

See the PostgreSQL documentation and RFC 5803 for details on this.

The passwords or secrets stored in the authentication file serve two purposes. First, they are used to verify the passwords of incoming client connections, if a password-based authentication method is configured. Second, they are used as the passwords for outgoing connections to the backend server, if the backend server requires password-based authentication (unless the password is specified directly in the database's connection string). The latter works if the password is stored in plain text or MD5-hashed. SCRAM secrets can only be used for logging into a server if the client authentication also uses SCRAM, the PgBouncer database definition does not specify a user name, and the SCRAM secrets are identical in PgBouncer and the PostgreSQL server (same salt and iterations, not merely the same password). This is due to an inherent security property of SCRAM: The stored SCRAM secret cannot by itself be used for deriving login credentials.

The authentication file can be written by hand, but it's also useful to generate it from some other list of users and passwords. See `./etc/mkauth.py` for a sample script to generate the authentication file from the `pg_shadow` system table. Alternatively, use

```
auth_query
```

instead of `auth_file` to avoid having to maintain a separate authentication file.\u0000

Configuring HBA-based Authentication for PgBouncer

PgBouncer supports HBA-based authentication. To configure HBA-based authentication for PgBouncer, you set `auth_type=hba` in the `pgbouncer.ini` configuration file. You also provide the filename of the HBA-format file in the `auth_hba_file` parameter of the `pgbouncer.ini` file.

Contents of an example PgBouncer HBA file named `hba_bouncer.conf`:

```
local       all     bouncer             trust
host        all     bouncer      127.0.0.1/32       trust
```

Example excerpt from the related `pgbouncer.ini` configuration file:

```
[databases]
p0 = port=15432 host=127.0.0.1 dbname=p0 user=bouncer pool_size=2
p1 = port=15432 host=127.0.0.1 dbname=p1 user=bouncer
...

[pgbouncer]
...
auth_type = hba
auth_file = userlist.txt
auth_hba_file = hba_bouncer.conf
...
```

Refer to the *HBA file format* discussion in the PgBouncer documentation for information about PgBouncer support of the HBA authentication file format.

Starting PgBouncer

You can run PgBouncer on the Greenplum Database master or on another server. If you install PgBouncer on a separate server, you can easily switch clients to the standby master by updating the PgBouncer configuration file and reloading the configuration using the PgBouncer Administration Console.

Follow these steps to set up PgBouncer.

1. Create a PgBouncer configuration file. For example, add the following text to a file named `pgbouncer.ini`:

   ```
   [databases]
   ```

```
postgres = host=127.0.0.1 port=5432 dbname=postgres
pgb_mydb = host=127.0.0.1 port=5432 dbname=mydb

[pgbouncer]
pool_mode = session
listen_port = 6543
listen_addr = 127.0.0.1
auth_type = md5
auth_file = users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = gpadmin
```

The file lists databases and their connection details. The file also configures the PgBouncer instance. Refer to the *pgbouncer.ini* reference page for information about the format and content of a PgBouncer configuration file.

2. Create an authentication file. The filename should be the name you specified for the `auth_file` parameter of the `pgbouncer.ini` file, `users.txt`. Each line contains a user name and password. The format of the password string matches the `auth_type` you configured in the PgBouncer configuration file. If the `auth_type` parameter is `plain`, the password string is a clear text password, for example:

   ```
   "gpadmin" "gpadmin1234"
   ```

   If the `auth_type` in the following example is `md5`, the authentication field must be MD5-encoded. The format for an MD5-encoded password is:

   ```
   "md5" + MD5_encoded(<password><username>)
   ```

3. Launch `pgbouncer`:

   ```
   $ $GPHOME/bin/pgbouncer -d pgbouncer.ini
   ```

   The `-d` option runs PgBouncer as a background (daemon) process. Refer to the *pgbouncer* reference page for the `pgbouncer` command syntax and options.

4. Update your client applications to connect to `pgbouncer` instead of directly to Greenplum Database server. For example, to connect to the Greenplum database named `mydb` configured above, run `psql` as follows:

   ```
   $ psql -p 6543 -U someuser pgb_mydb
   ```

   The `-p` option value is the `listen_port` that you configured for the PgBouncer instance.

Managing PgBouncer

PgBouncer provides a `psql`-like administration console. You log in to the PgBouncer Administration Console by specifying the PgBouncer port number and a virtual database named `pgbouncer`. The console accepts SQL-like commands that you can use to monitor, reconfigure, and manage PgBouncer.

For complete documentation of PgBouncer Administration Console commands, refer to the *PgBouncer Administration Console* command reference.

Follow these steps to get started with the PgBouncer Administration Console.

1. Use `psql` to log in to the `pgbouncer` virtual database:

   ```
   $ psql -p 6543 -U username pgbouncer
   ```

The *username* that you specify must be listed in the `admin_users` parameter in the `pgbouncer.ini` configuration file. You can also log in to the PgBouncer Administration Console with the current Unix username if the `pgbouncer` process is running under that user's UID.

2. To view the available PgBouncer Administration Console commands, run the `SHOW help` command:

```
pgbouncer=# SHOW help;
NOTICE:  Console usage
DETAIL:
    SHOW HELP|CONFIG|DATABASES|POOLS|CLIENTS|SERVERS|VERSION
    SHOW FDS|SOCKETS|ACTIVE_SOCKETS|LISTS|MEM
    SHOW DNS_HOSTS|DNS_ZONES
    SHOW STATS|STATS_TOTALS|STATS_AVERAGES
    SET key = arg
    RELOAD
    PAUSE [<db>]
    RESUME [<db>]
    DISABLE <db>
    ENABLE <db>
    KILL <db>
    SUSPEND
    SHUTDOWN
```

3. If you update PgBouncer configuration by editing the `pgbouncer.ini` configuration file, you use the `RELOAD` command to reload the file:

```
pgbouncer=# RELOAD;
```

Mapping PgBouncer Clients to Greenplum Database Server Connections

To map a PgBouncer client to a Greenplum Database server connection, use the PgBouncer Administration Console `SHOW CLIENTS` and `SHOW SERVERS` commands:

1. Use `ptr` and `link` to map the local client connection to the server connection.
2. Use the `addr` and the `port` of the client connection to identify the TCP connection from the client.
3. Use `local_addr` and `local_port` to identify the TCP connection to the server.

Database Application Interfaces

You may want to develop your own client applications that interface to Greenplum Database. PostgreSQL provides a number of database drivers for the most commonly used database application programming interfaces (APIs), which can also be used with Greenplum Database. These drivers are available as a separate download. Each driver (except libpq, which comes with PostgreSQL) is an independent PostgreSQL development project and must be downloaded, installed and configured to connect to Greenplum Database. The following drivers are available:

Table 19: Greenplum Database Interfaces

API	PostgreSQL Driver	Download Link
ODBC	Greenplum DataDirect ODBC Driver	*https://network.pivotal.io/products/pivotal-gpdb*.
JDBC	Greenplum DataDirect JDBC Driver	*https://network.pivotal.io/products/pivotal-gpdb*

API	PostgreSQL Driver	Download Link
Perl DBI	pgperl	*https://metacpan.org/release/DBD-Pg*
Python DBI	pygresql	*http://www.pygresql.org/*
libpq C Library	libpq	*https://www.postgresql.org/docs/9.4/libpq.html*

General instructions for accessing a Greenplum Database with an API are:

1. Download your programming language platform and respective API from the appropriate source. For example, you can get the Java Development Kit (JDK) and JDBC API from Oracle.
2. Write your client application according to the API specifications. When programming your application, be aware of the SQL support in Greenplum Database so you do not include any unsupported SQL syntax.

   See the *Greenplum Database Reference Guide* for more information.

Download the appropriate driver and configure connectivity to your Greenplum Database master instance.

Troubleshooting Connection Problems

A number of things can prevent a client application from successfully connecting to Greenplum Database. This topic explains some of the common causes of connection problems and how to correct them.

Table 20: Common connection problems

Problem	Solution
No `pg_hba.conf` entry for host or user	To enable Greenplum Database to accept remote client connections, you must configure your Greenplum Database master instance so that connections are allowed from the client hosts and database users that will be connecting to Greenplum Database. This is done by adding the appropriate entries to the `pg_hba.conf` configuration file (located in the master instance's data directory). For more detailed information, see *Allowing Connections to Greenplum Database*.
Greenplum Database is not running	If the Greenplum Database master instance is down, users will not be able to connect. You can verify that the Greenplum Database system is up by running the `gpstate` utility on the Greenplum master host.
Network problems Interconnect timeouts	If users connect to the Greenplum master host from a remote client, network problems can prevent a connection (for example, DNS host name resolution problems, the host system is down, and so on.). To ensure that network problems are not the cause, connect to the Greenplum master host from the remote client host. For example: `ping` *`hostname`* . If the system cannot resolve the host names and IP addresses of the hosts involved in Greenplum Database, queries and connections will fail. For some operations, connections to the Greenplum Database master use `localhost` and others use the actual host name, so you must be able to resolve both. If you encounter this error, first make sure you can connect to each host in your Greenplum Database array from the master host over the network. In the `/etc/hosts` file of the master and all segments, make sure you have the correct host names and IP addresses for all hosts involved in the Greenplum Database array. The `127.0.0.1` IP must resolve to `localhost`.

Problem	Solution
Too many clients already	By default, Greenplum Database is configured to allow a maximum of 250 concurrent user connections on the master and 750 on a segment. A connection attempt that causes that limit to be exceeded will be refused. This limit is controlled by the `max_connections` parameter in the `postgresql.conf` configuration file of the Greenplum Database master. If you change this setting for the master, you must also make appropriate changes at the segments.

Configuring the Greenplum Database System

Server configuration parameters affect the behavior of Greenplum Database. They are part of the PostgreSQL "Grand Unified Configuration" system, so they are sometimes called "GUCs." Most of the Greenplum Database server configuration parameters are the same as the PostgreSQL configuration parameters, but some are Greenplum-specific.

About Greenplum Database Master and Local Parameters

Server configuration files contain parameters that configure server behavior. The Greenplum Database configuration file, `postgresql.conf`, resides in the data directory of the database instance.

The master and each segment instance have their own `postgresql.conf` file. Some parameters are *local*: each segment instance examines its `postgresql.conf` file to get the value of that parameter. Set local parameters on the master and on each segment instance.

Other parameters are *master* parameters that you set on the master instance. The value is passed down to (or in some cases ignored by) the segment instances at query run time.

See the *Greenplum Database Reference Guide* for information about *local* and *master* server configuration parameters.

Setting Configuration Parameters

Many configuration parameters limit who can change them and where or when they can be set. For example, to change certain parameters, you must be a Greenplum Database superuser. Other parameters can be set only at the system level in the `postgresql.conf` file or require a system restart to take effect.

Many configuration parameters are *session* parameters. You can set session parameters at the system level, the database level, the role level or the session level. Database users can change most session parameters within their session, but some require superuser permissions.

See the *Greenplum Database Reference Guide* for information about setting server configuration parameters.

Setting a Local Configuration Parameter

To change a local configuration parameter across multiple segments, update the parameter in the `postgresql.conf` file of each targeted segment, both primary and mirror. Use the `gpconfig` utility to set a parameter in all Greenplum `postgresql.conf` files. For example:

```
$ gpconfig -c gp_vmem_protect_limit -v 4096
```

Restart Greenplum Database to make the configuration changes effective:

```
$ gpstop -r
```

Setting a Master Configuration Parameter

To set a master configuration parameter, set it at the Greenplum Database master instance. If it is also a *session* parameter, you can set the parameter for a particular database, role or session. If a parameter is set at multiple levels, the most granular level takes precedence. For example, session overrides role, role overrides database, and database overrides system.

Setting Parameters at the System Level

Master parameter settings in the master `postgresql.conf` file are the system-wide default. To set a master parameter:

1. Edit the `$MASTER_DATA_DIRECTORY/postgresql.conf` file.
2. Find the parameter to set, uncomment it (remove the preceding # character), and type the desired value.
3. Save and close the file.
4. For *session* parameters that do not require a server restart, upload the `postgresql.conf` changes as follows:

   ```
   $ gpstop -u
   ```

5. For parameter changes that require a server restart, restart Greenplum Database as follows:

   ```
   $ gpstop -r
   ```

For details about the server configuration parameters, see the *Greenplum Database Reference Guide*.

Setting Parameters at the Database Level

Use `ALTER DATABASE` to set parameters at the database level. For example:

```
=# ALTER DATABASE mydatabase SET search_path TO myschema;
```

When you set a session parameter at the database level, every session that connects to that database uses that parameter setting. Settings at the database level override settings at the system level.

Setting Parameters at the Role Level

Use `ALTER ROLE` to set a parameter at the role level. For example:

```
=# ALTER ROLE bob SET search_path TO bobschema;
```

When you set a session parameter at the role level, every session initiated by that role uses that parameter setting. Settings at the role level override settings at the database level.

Setting Parameters in a Session

Any session parameter can be set in an active database session using the `SET` command. For example:

```
=# SET statement_mem TO '200MB';
```

The parameter setting is valid for the rest of that session or until you issue a `RESET` command. For example:

```
=# RESET statement_mem;
```

Settings at the session level override those at the role level.

Viewing Server Configuration Parameter Settings

The SQL command `SHOW` allows you to see the current server configuration parameter settings. For example, to see the settings for all parameters:

```
$ psql -c 'SHOW ALL;'
```

`SHOW` lists the settings for the master instance only. To see the value of a particular parameter across the entire system (master and all segments), use the `gpconfig` utility. For example:

```
$ gpconfig --show max_connections
```

Configuration Parameter Categories

Configuration parameters affect categories of server behaviors, such as resource consumption, query tuning, and authentication. Refer to *Parameter Categories* in the *Greenplum Database Reference Guide* for a list of Greenplum server configuration parameter categories.

Enabling Compression

You can configure Greenplum Database to use data compression with some database features and with some utilities. Compression reduces disk usage and improves I/O across the system, however, it adds some performance overhead when compressing and decompressing data.

You can configure support for data compression with these features and utilities. See the specific feature or utility for information about support for compression.

- Append-optimized tables support compressing table data. See *CREATE TABLE*.
- User-defined data types can be defined to compress data. See *CREATE TYPE*.
- The external table protocols *gpfdist* (*gpfdists*), *s3*, and *pxf* support compression when accessing external data. For information about external tables, see *CREATE EXTERNAL TABLE*.
- Workfiles (temporary spill files that are created when executing a query that requires more memory than it is allocated) can be compressed. See the server configuration parameter *gp_workfile_compression*.
- The Greenplum Database utilities *gpbackup*, *gprestore*, *gpcopy*, *gpload*, and *gplogfilter* support compression.

For some compression algorithms (such as zlib) Greenplum Database requires software packages installed on the host system. For information about required software packages, see the *Greenplum Database Installation Guide*.

Configuring Proxies for the Greenplum Interconnect

You can configure a Greenplum system to use proxies for interconnect communication to reduce the use of connections and ports during query processing.

The Greenplum *interconnect* (the networking layer) refers to the inter-process communication between segments and the network infrastructure on which this communication relies. For information about the Greenplum architecture and interconnect, see *About the Greenplum Architecture*.

In general, when running a query, a QD (query dispatcher) on the Greenplum master creates connections to one or more QE (query executor) processes on segments, and a QE can create connections to other QEs. For a description of Greenplum query processing and parallel query processing, see *About Greenplum Query Processing*.

By default, connections between the QD on the master and QEs on segment instances and between QEs on different segment instances require a separate network port. You can configure a Greenplum system to use proxies when Greenplum communicates between the QD and QEs and between QEs on different

segment instances. The interconnect proxies require only one network connection for Greenplum internal communication between two segment instances, so it consumes fewer connections and ports than `TCP` mode, and has better performance than `UDPIFC` mode in a high-latency network.

To enable interconnect proxies for the Greenplum system, set these system configuration parameters.

- List the proxy ports with the parameter *`gp_interconnect_proxy_addresses`*. You must specify a proxy port for the master, standby master, and all segment instances.
- Set the parameter *`gp_interconnect_type`* to `proxy`.

  Note: When expanding a Greenplum Database system, you must disable interconnect proxies before adding new hosts and segment instances to the system, and you must update the `gp_interconnect_proxy_addresses` parameter with the newly-added segment instances before you re-enable interconnect proxies.

Example

This example sets up a Greenplum system to use proxies for the Greenplum interconnect when running queries. The example sets the *`gp_interconnect_proxy_addresses`* parameter and tests the proxies before setting the *`gp_interconnect_type`* parameter for the Greenplum system.

- *Setting the Interconnect Proxy Addresses*
- *Testing the Interconnect Proxies*
- *Setting Interconnect Proxies for the System*

Setting the Interconnect Proxy Addresses

Set the `gp_interconnect_proxy_addresses` parameter to specify the proxy ports for the master and segment instances. The syntax for the value has the following format and you must specify the parameter value as a single-quoted string.

```
<db_id>:<cont_id>:<seg_ip>:<port>[,<dbid>:<segid>:<ip>:<port> ... ]
```

For the master, standby master, and segment instance, the first three fields, *db_id*, *cont_id*, and *seg_ip* can be found in the *`gp_segment_configuration`* catalog table. The fourth field, *port*, is the proxy port for the Greenplum master or a segment instance.

- *db_id* is the `dbid` column in the catalog table.
- *cont_id* is the `content` column in the catalog table.
- *seg_ip* is the IP address corresponding to the `address` column in the catalog table. If the `address` is a hostname, use the IP address of the hostname.
- *port* is the TCP/IP port for the segment instance proxy that you specify.

This is an example PL/Python function that displays or sets the segment instance proxy port values for the `gp_interconnect_proxy_addresses` parameter. To create and run the function, you must enable PL/Python in the database with the `CREATE EXTENSION plpythonu` command.

```
--
-- A PL/Python function to setup the interconnect proxy addresses.
-- Requires the Python modules os and socket.
--
-- Usage:
--   select my_setup_ic_proxy(-1000, '');              -- display IC proxy
 values for segments
--   select my_setup_ic_proxy(-1000, 'update proxy');  -- update the
 gp_interconnect_proxy_addresses parameter
--
-- The first argument, "delta", is used to calculate the proxy port with
 this formula:
--
```

```
--   proxy_port = postmaster_port + delta
--
-- The second argument, "action", is used to update the
 gp_interconnect_proxy_addresses parameter.
-- The parameter is not updated unless "action" is 'update proxy'.
-- Note that running  "gpstop -u" is required for the update to take
 effect.
-- A Greenplum system restart will also work.
--
create or replace function my_setup_ic_proxy(delta int, action text)
returns table(dbid smallint, content smallint, ip text, port int) as $$
    import os
    import socket

    results = []
    value = ''

    segs = plpy.execute('''SELECT dbid, content, port, address
                             FROM gp_segment_configuration
                            ORDER BY 1''')
    for seg in segs:
        dbid = seg['dbid']
        content = seg['content']
        port = seg['port']
        address = seg['address']

        # lookup ip of the address
        ip = socket.gethostbyname(address)

        # decide the proxy port
        port = port + delta

        # append to the result list
        results.append((dbid, content, ip, port))

        # build the value for the GUC
        if value:
            value += ','
        value += '{}:{}:{}:{}'.format(dbid, content, ip, port)

    if action.lower() == 'update proxy':
        os.system('''gpconfig --skipvalidation -c
 gp_interconnect_proxy_addresses -v "'{}'"'''.format(value))
        plpy.notice('''the settings are applied, please reload with 'gpstop
 -u' to take effect.''')
    else:
        plpy.notice('''if the settings are correct, re-run with 'update
 proxy' to apply.''')
    return results
$$ language plpythonu execute on master;
```

Note: When you run the function, you should connect to the database using the Greenplum interconnect type `UDPIFC` or `TCP`. This example uses `psql` to connect to the database `mytest` with the interconnect type `UDPIFC`.

```
PGOPTIONS="-c gp_interconnect_type=udpifc" psql -d mytest
```

Running this command lists the segment instance values for the `gp_interconnect_proxy_addresses` parameter.

```
select my_setup_ic_proxy(-1000, '');
```

This command runs the function to set the parameter.

```
select my_setup_ic_proxy(-1000, 'update proxy');
```

As an alternative, you can run the *gpconfig* utility to set the `gp_interconnect_proxy_addresses` parameter. To set the value as a string, the value is a single-quoted string that is enclosed in double quotes. The example Greenplum system consists of a master and a single segment instance.

```
gpconfig --skipvalidation -c gp_interconnect_proxy_addresses -v
 "'1:-1:192.168.180.50:35432,2:0:192.168.180.54:35000'"
```

After setting the `gp_interconnect_proxy_addresses` parameter, reload the `postgresql.conf` file with the `gpstop -u` command. This command does not stop and restart the Greenplum system.

Testing the Interconnect Proxies

To test the proxy ports configured for the system, you can set the `PGOPTIONS` environment variable when you start a `psql` session in a command shell. This command sets the environment variable to enable interconnect proxies, starts `psql`, and logs into the database `mytest`.

```
PGOPTIONS="-c gp_interconnect_type=proxy" psql -d mytest
```

You can run queries in the shell to test the system. For example, you can run a query that accesses all the primary segment instances. This query displays the segment IDs and number of rows on the segment instance from the table `sales`.

```
# SELECT gp_segment_id, COUNT(*) FROM sales GROUP BY gp_segment_id ;
```

Setting Interconnect Proxies for the System

After you have tested the interconnect proxies for the system, set the server configuration parameter for the system with the `gpconfig` utility.

```
gpconfig -c  gp_interconnect_type -v proxy
```

Reload the `postgresql.conf` file with the `gpstop -u` command. This command does not stop and restart the Greenplum system.

Enabling High Availability and Data Consistency Features

The fault tolerance and the high-availability features of Greenplum Database can be configured.

> **Important:** When data loss is not acceptable for a Greenplum Database cluster, Greenplum master and segment mirroring is recommended. If mirroring is not enabled then Greenplum stores only one copy of the data, so the underlying storage media provides the only guarantee for data availability and correctness in the event of a hardware failure.
>
> Kubernetes enables quick recovery from both pod and host failures, and Kubernetes storage services provide a high level of availability for the underlying data. Furthermore, virtualized environments make it difficult to ensure the anti-affinity guarantees required for Greenplum mirroring solutions. For these reasons, mirrorless deployments are fully supported with Greenplum for Kubernetes. Other deployment environments are generally not supported for production use unless both Greenplum master and segment mirroring are enabled.

For information about the utilities that are used to enable high availability, see the *Greenplum Database Utility Guide*.

Overview of Greenplum Database High Availability

A Greenplum Database system can be made highly available by providing a fault-tolerant hardware platform, by enabling Greenplum Database high-availability features, and by performing regular monitoring and maintenance procedures to ensure the health of all system components.

Hardware components will eventually fail, whether due to normal wear or an unexpected circumstance. Loss of power can lead to temporarily unavailable components. A system can be made highly available by providing redundant standbys for components that can fail so that services can continue uninterrupted when a failure does occur. In some cases, the cost of redundancy is higher than users' tolerance for interruption in service. When this is the case, the goal is to ensure that full service is able to be restored, and can be restored within an expected timeframe.

With Greenplum Database, fault tolerance and data availability is achieved with:

- *Hardware level RAID storage protection*
- *Data storage checksums*
- *Greenplum segment mirroring*
- *Master mirroring*
- *Dual clusters*
- *Database backup and restore*

Hardware level RAID

A best practice Greenplum Database deployment uses hardware level RAID to provide high performance redundancy for single disk failure without having to go into the database level fault tolerance. This provides a lower level of redundancy at the disk level.

Data storage checksums

Greenplum Database uses checksums to verify that data loaded from disk to memory has not been corrupted on the file system.

Greenplum Database has two kinds of storage for user data: heap and append-optimized. Both storage models use checksums to verify data read from the file system and, with the default settings, they handle checksum verification errors in a similar way.

Greenplum Database master and segment database processes update data on pages in the memory they manage. When a memory page is updated and flushed to disk, checksums are computed and saved with the page. When a page is later retrieved from disk, the checksums are verified and the page is only permitted to enter managed memory if the verification succeeds. A failed checksum verification is an indication of corruption in the file system and causes Greenplum Database to generate an error, aborting the transaction.

The default checksum settings provide the best level of protection from undetected disk corruption propagating into the database and to mirror segments.

Heap checksum support is enabled by default when the Greenplum Database cluster is initialized with the `gpinitsystem` management utility. Although it is strongly discouraged, a cluster can be initialized without heap checksum support by setting the `HEAP_CHECKSUM` parameter to off in the `gpinitsystem` cluster configuration file. See *`gpinitsystem`*.

Once initialized, it is not possible to change heap checksum support for a cluster without reinitializing the system and reloading databases.

You can check the read-only server configuration parameter *`data_checksums`* to see if heap checksums are enabled in a cluster:

```
$ gpconfig -s data_checksums
```

When a Greenplum Database cluster starts up, the `gpstart` utility checks that heap checksums are consistently enabled or disabled on the master and all segments. If there are any differences, the cluster fails to start. See *`gpstart`*.

In cases where it is necessary to ignore heap checksum verification errors so that data can be recovered, setting the *`ignore_checksum_failure`* system configuration parameter to on causes Greenplum Database to issue a warning when a heap checksum verification fails, but the page is then permitted to load into managed memory. If the page is updated and saved to disk, the corrupted data could be replicated to the mirror segment. Because this can lead to data loss, setting `ignore_checksum_failure` to on should only be done to enable data recovery.

For append-optimized storage, checksum support is one of several storage options set at the time an append-optimized table is created with the `CREATE TABLE` command. The default storage options are specified in the `gp_default_storage_options` server configuration parameter. The `checksum` storage option is enabled by default and disabling it is strongly discouraged.

If you choose to disable checksums for an append-optimized table, you can either

- change the `gp_default_storage_options` configuration parameter to include `checksum=false` before creating the table, or
- add the `checksum=false` option to the `WITH` *`storage_options`* clause of the `CREATE TABLE` statement.

Note that the `CREATE TABLE` statement allows you to set storage options, including checksums, for individual partition files.

See the *`CREATE TABLE`* command reference and the *`gp_default_storage_options`* configuration parameter reference for syntax and examples.

Segment Mirroring

Greenplum Database stores data in multiple segment instances, each of which is a Greenplum Database PostgreSQL instance. The data for each table is spread between the segments based on the distribution policy that is defined for the table in the DDL at the time the table is created. When segment mirroring is enabled, for each segment instance there is a *primary* and *mirror* pair. The mirror segment is kept up to date with the primary segment using Write-Ahead Logging (WAL)-based streaming replication. See *Overview of Segment Mirroring*.

The mirror instance for each segment is usually initialized with the `gpinitsystem` utility or the `gpexpand` utility. As a best practice, the mirror runs on a different host than the primary instance to protect from a single machine failure. There are different strategies for assigning mirrors to hosts. When choosing the layout of the primaries and mirrors, it is important to consider the failure scenarios to ensure that processing skew is minimized in the case of a single machine failure.

Master Mirroring

There are two master instances in a highly available cluster, a *primary* and a *standby*. As with segments, the master and standby should be deployed on different hosts so that the cluster can tolerate a single host failure. Clients connect to the primary master and queries can be executed only on the primary master. The standby master is kept up to date with the primary master using Write-Ahead Logging (WAL)-based streaming replication. See *Overview of Master Mirroring*.

If the master fails, the administrator runs the `gpactivatestandby` utility to have the standby master take over as the new primary master. You can configure a virtual IP address for the master and standby so that client programs do not have to switch to a different network address when the current master changes. If the master host fails, the virtual IP address can be swapped to the actual acting master.

Dual Clusters

An additional level of redundancy can be provided by maintaining two Greenplum Database clusters, both storing the same data.

Two methods for keeping data synchronized on dual clusters are "dual ETL" and "backup/restore."

Dual ETL provides a complete standby cluster with the same data as the primary cluster. ETL (extract, transform, and load) refers to the process of cleansing, transforming, validating, and loading incoming data into a data warehouse. With dual ETL, this process is executed twice in parallel, once on each cluster, and is validated each time. It also allows data to be queried on both clusters, doubling the query throughput. Applications can take advantage of both clusters and also ensure that the ETL is successful and validated on both clusters.

To maintain a dual cluster with the backup/restore method, create backups of the primary cluster and restore them on the secondary cluster. This method takes longer to synchronize data on the secondary cluster than the dual ETL strategy, but requires less application logic to be developed. Populating a second cluster with backups is ideal in use cases where data modifications and ETL are performed daily or less frequently.

Backup and Restore

Making regular backups of the databases is recommended except in cases where the database can be easily regenerated from the source data. Backups should be taken to protect from operational, software, and hardware errors.

Use the *gpbackup* utility to backup Greenplum databases. `gpbackup` performs the backup in parallel across segments, so backup performance scales up as hardware is added to the cluster.

When designing a backup strategy, a primary concern is where to store the backup data. The data each segment manages can be backed up on the segment's local storage, but should not be stored there permanently—the backup reduces disk space available to the segment and, more importantly, a hardware failure could simultaneously destroy the segment's live data and the backup. After performing a backup, the backup files should be moved from the primary cluster to separate, safe storage. Alternatively, the backup can be made directly to separate storage.

Using a Greenplum Database storage plugin with the `gpbackup` and `gprestore` utilities, you can send a backup to, or retrieve a backup from a remote location or a storage appliance. Greenplum Database storage plugins support connecting to locations including Amazon Simple Storage Service (Amazon S3) locations and Dell EMC Data Domain storage appliances.

Using the Backup/Restore Storage Plugin API you can create a custom plugin that the `gpbackup` and `gprestore` utilities can use to integrate a custom backup storage system with the Greenplum Database.

For information about using `gpbackup` and `gprestore`, see *Parallel Backup with gpbackup and gprestore*.

Overview of Segment Mirroring

When Greenplum Database High Availability is enabled, there are two types of segment instances: *primary* and *mirror*. Each primary segment has one corresponding mirror segment. A primary segment instance receives requests from the master to make changes to the segment data and then replicates those changes to the corresponding mirror. If Greenplum Database detects that a primary segment has failed or become unavailable, it changes the role of its mirror segment to primary segment and the role of the unavailable primary segment to mirror segment. Transactions in progress when the failure occurred roll back and must be restarted. The administrator must then recover the mirror segment, allow the mirror to synchronize with the current primary segment, and then exchange the primary and mirror segments so they are in their preferred roles.

If segment mirroring is not enabled, the Greenplum Database system shuts down if a segment instance fails. Administrators must manually recover all failed segments before Greenplum Database operations can resume.

When segment mirroring is enabled for an existing system, the primary segment instances continue to provide service to users while a snapshot of the primary segments are taken. While the snapshots are taken and deployed on the mirror segment instances, changes to the primary segment are also recorded.

After the snapshot has been deployed on the mirror segment, the mirror segment is synchronized and kept current using Write-Ahead Logging (WAL)-based streaming replication. Greenplum Database WAL replication uses the `walsender` and `walreceiver` replication processes. The `walsender` process is a primary segment process. The `walreceiver` is a mirror segment process.

When database changes occur, the logs that capture the changes are streamed to the mirror segment to keep it current with the corresponding primary segments. During WAL replication, database changes are written to the logs before being applied, to ensure data integrity for any in-process operations.

When Greenplum Database detects a primary segment failure, the WAL replication process stops and the mirror segment automatically starts as the active primary segment. If a mirror segment fails or becomes inaccessible while the primary is active, the primary segment tracks database changes in logs that are applied to the mirror when it is recovered. For information about segment fault detection and the recovery process, see *Detecting a Failed Segment*.

These Greenplum Database system catalog tables contain mirroring and replication information.

- The catalog table *gp_segment_configuration* contains the current configuration and state of primary and mirror segment instances and the master and standby master instance.
- The catalog view *gp_stat_replication* contains replication statistics of the `walsender` processes that are used for Greenplum Database master and segment mirroring.

About Segment Mirroring Configurations

Mirror segment instances can be placed on hosts in the cluster in different configurations. As a best practice, a primary segment and the corresponding mirror are placed on different hosts. Each host must have the same number of primary and mirror segments. When you create segment mirrors with the Greenplum Database utilities *gpinitsystem* or *gpaddmirrors* you can specify the segment mirror configuration, group mirroring (the default) or spread mirroring. With `gpaddmirrors`, you can create custom mirroring configurations with a `gpaddmirrors` configuration file and specify the file on the command line.

Group mirroring is the default mirroring configuration when you enable mirroring during system initialization. The mirror segments for each host's primary segments are placed on one other host. If a single host fails, the number of active primary segments doubles on the host that backs the failed host. *Figure 15: Group Segment Mirroring in Greenplum Database* illustrates a group mirroring configuration.

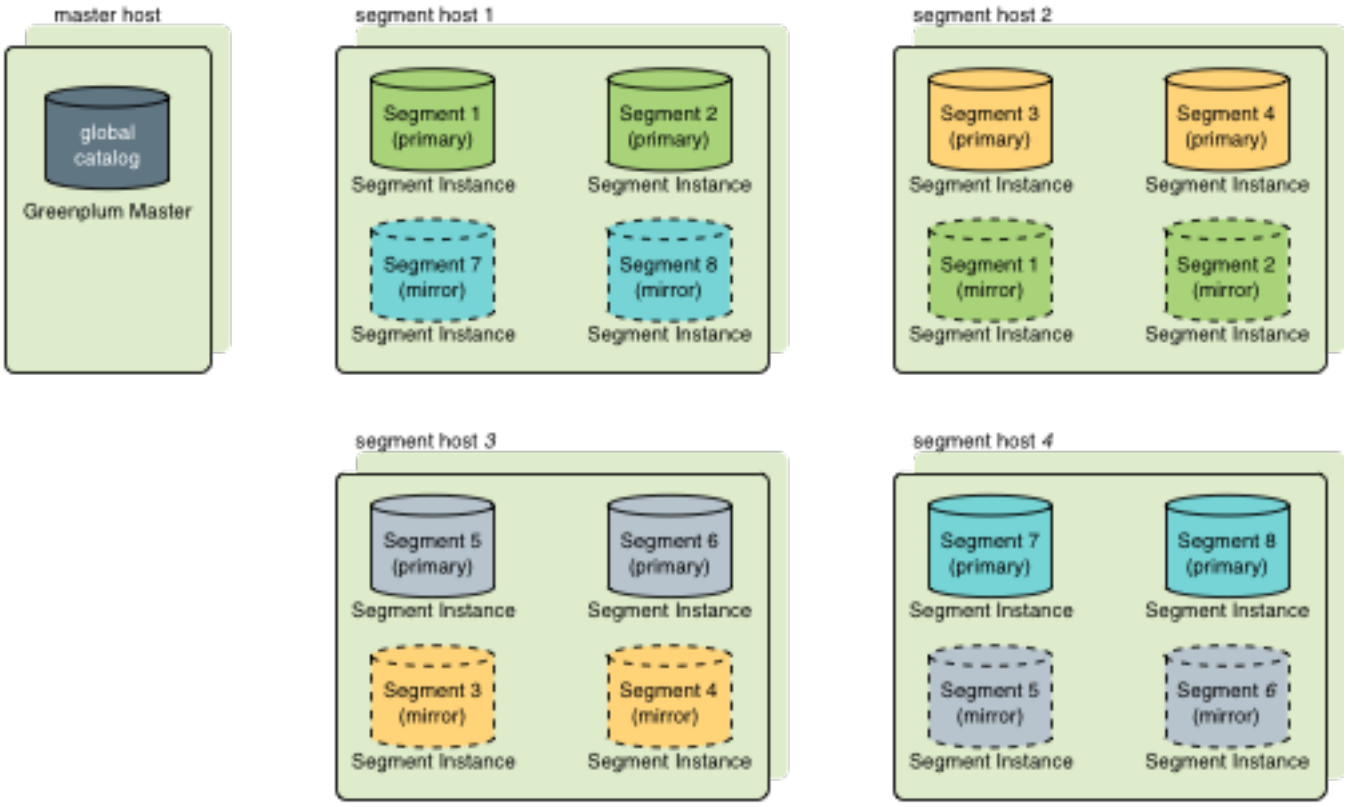


Figure 15: Group Segment Mirroring in Greenplum Database

Spread mirroring can be specified during system initialization. This configuration spreads each host's mirrors over multiple hosts so that if any single host fails, no other host will have more than one mirror promoted to an active primary segment. Spread mirroring is possible only if there are more hosts than segments per host. *Figure 16: Spread Segment Mirroring in Greenplum Database* illustrates the placement of mirrors in a spread segment mirroring configuration.

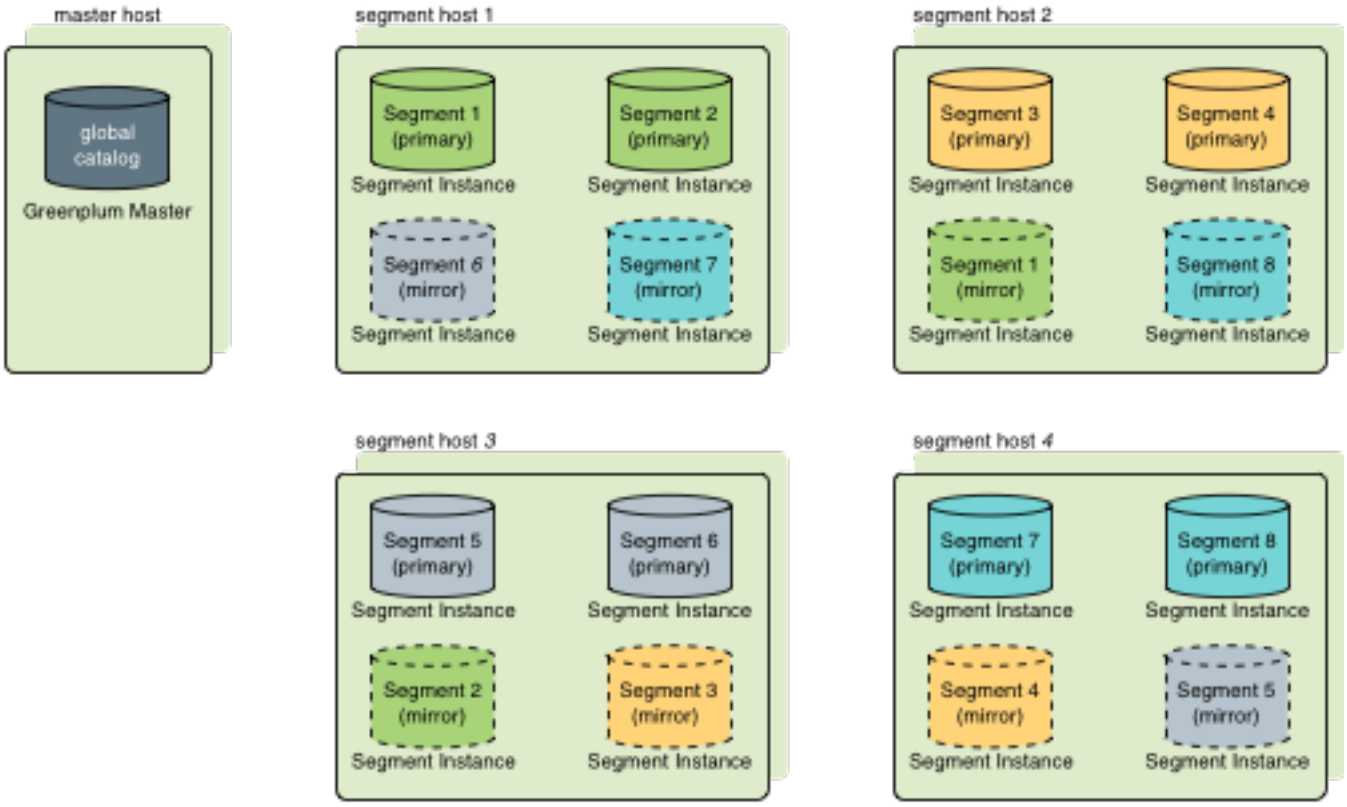


Figure 16: Spread Segment Mirroring in Greenplum Database

> **Note:** You must ensure you have the appropriate number of host systems for your mirroring configuration when you create a system or when you expand a system. For example, to create a system that is configured with spread mirroring requires more hosts than segment instances per host, and a system that is configured with group mirroring requires at least two new hosts when expanding the system. For information about segment mirroring configurations, see *Segment Mirroring Configurations*. For information about expanding systems with segment mirroring enabled, see *Planning Mirror Segments*.

Overview of Master Mirroring

You can deploy a backup or mirror of the master instance on a separate host machine. The backup master instance, called the *standby master*, serves as a warm standby if the primary master becomes nonoperational. You create a standby master from the primary master while the primary is online.

When you enable master mirroring for an existing system, the primary master continues to provide service to users while a snapshot of the primary master instance is taken. While the snapshot is taken and deployed on the standby master, changes to the primary master are also recorded. After the snapshot has been deployed on the standby master, the standby master is synchronized and kept current using Write-Ahead Logging (WAL)-based streaming replication. Greenplum Database WAL replication uses the `walsender` and `walreceiver` replication processes. The `walsender` process is a primary master process. The `walreceiver` is a standby master process.

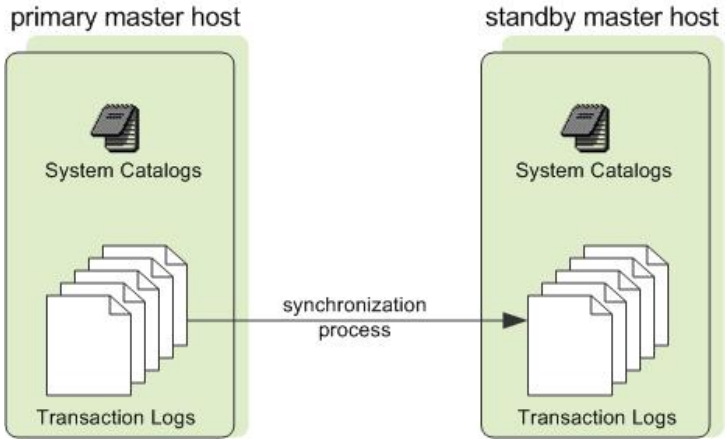


Figure 17: Master Mirroring in Greenplum Database

Since the master does not house user data, only system catalog tables are synchronized between the primary and standby masters. When these tables are updated, the replication logs that capture the changes are streamed to the standby master to keep it current with the primary. During WAL replication, all database modifications are written to replication logs before being applied, to ensure data integrity for any in-process operations.

This is how Greenplum Database handles a master failure.

- If the primary master fails, the Greenplum Database system shuts down and the master replication process stops. The administrator runs the `gpactivatestandby` utility to have the standby master take over as the new primary master. Upon activation of the standby master, the replicated logs reconstruct the state of the primary master at the time of the last successfully committed transaction. The activated standby master then functions as the Greenplum Database master, accepting connections on the port specified when standby master was initialized. See *Recovering a Failed Master*.
- If the standby master fails or becomes inaccessible while the primary master is active, the primary master tracks database changes in logs that are applied to the standby master when it is recovered.

These Greenplum Database system catalog tables contain mirroring and replication information.

- The catalog table *`gp_segment_configuration`* contains the current configuration and state of primary and mirror segment instances and the master and standby master instance.
- The catalog view *`gp_stat_replication`* contains replication statistics of the `walsender` processes that are used for Greenplum Database master and segment mirroring.

Enabling Mirroring in Greenplum Database

You can configure your Greenplum Database system with mirroring at setup time using `gpinitsystem` or enable mirroring later using `gpaddmirrors` and `gpinitstandby`. This topic assumes you are adding mirrors to an existing system that was initialized without mirrors.

You can enable the following types of mirroring:

- *Enabling Segment Mirroring*
- *Enabling Master Mirroring*

Enabling Segment Mirroring

Mirror segments allow database queries to fail over to a backup segment if the primary segment is unavailable. By default, mirrors are configured on the same array of hosts as the primary segments. You may choose a completely different set of hosts for your mirror segments so they do not share machines with any of your primary segments.

Important: During the online data replication process, Greenplum Database should be in a quiescent state, workloads and other queries should not be running.

To add segment mirrors to an existing system (same hosts as primaries)

1. Allocate the data storage area for mirror data on all segment hosts. The data storage area must be different from your primary segments' file system location.
2. Use *gpssh-exkeys* to ensure that the segment hosts can SSH and SCP to each other without a password prompt.
3. Run the *gpaddmirrors* utility to enable mirroring in your Greenplum Database system. For example, to add 10000 to your primary segment port numbers to calculate the mirror segment port numbers:

```
$ gpaddmirrors -p 10000
```

Where `-p` specifies the number to add to your primary segment port numbers. Mirrors are added with the default group mirroring configuration.

To add segment mirrors to an existing system (different hosts from primaries)

1. Ensure the Greenplum Database software is installed on all hosts. See the *Greenplum Database Installation Guide* for detailed installation instructions.
2. Allocate the data storage area for mirror data, and tablespaces if needed, on all segment hosts.
3. Use `gpssh-exkeys` to ensure the segment hosts can SSH and SCP to each other without a password prompt.
4. Create a configuration file that lists the host names, ports, and data directories on which to create mirrors. To create a sample configuration file to use as a starting point, run:

```
$ gpaddmirrors -o filename
```

The format of the mirror configuration file is:

```
row_id=contentID|address|port|data_dir
```

Where `row_id` is the row in the file, *contentID* is the segment instance content ID, *address* is the host name or IP address of the segment host, *port* is the communication port, and `data_dir` is the segment instance data directory.

For example, this is contents of a mirror configuration file for two segment hosts and two segment instances per host:

```
0=2|sdw1-1|41000|/data/mirror1/gp2
1=3|sdw1-2|41001|/data/mirror2/gp3
2=0|sdw2-1|41000|/data/mirror1/gp0
3=1|sdw2-2|41001|/data/mirror2/gp1
```

5. Run the `gpaddmirrors` utility to enable mirroring in your Greenplum Database system:

```
$ gpaddmirrors -i mirror_config_file
```

The `-i` option specifies the mirror configuration file you created.

Enabling Master Mirroring

You can configure a new Greenplum Database system with a standby master using `gpinitsystem` or enable it later using `gpinitstandby`. This topic assumes you are adding a standby master to an existing system that was initialized without one.

For information about the utilities *gpinitsystem* and *gpinitstandby*, see the *Greenplum Database Utility Guide*.

To add a standby master to an existing system

1. Ensure the standby master host is installed and configured: `gpadmin` system user created, Greenplum Database binaries installed, environment variables set, SSH keys exchanged, and that the data directories and tablespace directories, if needed, are created.
2. Run the `gpinitstandby` utility on the currently active *primary* master host to add a standby master host to your Greenplum Database system. For example:

   ```
   $ gpinitstandby -s smdw
   ```

   Where `-s` specifies the standby master host name.

To switch operations to a standby master, see *Recovering a Failed Master*.

To check the status of the master mirroring process (optional)

You can run the `gpstate` utility with the `-f` option to display details of the standby master host.

```
$ gpstate -f
```

The standby master status should be passive, and the WAL sender state should be streaming.

For information about the *gpstate* utility, see the *Greenplum Database Utility Guide*.

Detecting a Failed Segment

With segment mirroring enabled, Greenplum Database automatically fails over to a mirror segment instance when a primary segment instance goes down. Provided one segment instance is online per portion of data, users may not realize a segment is down. If a transaction is in progress when a fault occurs, the in-progress transaction rolls back and restarts automatically on the reconfigured set of segments. The *gpstate* utility can be used to identify failed segments. The utility displays information from the catalog tables including *gp_segment_configuration*.

If the entire Greenplum Database system becomes nonoperational due to a segment failure (for example, if mirroring is not enabled or not enough segments are online to access all user data), users will see errors when trying to connect to a database. The errors returned to the client program may indicate the failure. For example:

```
ERROR: All segment databases are unavailable
```

How a Segment Failure is Detected and Managed

On the Greenplum Database master host, the Postgres `postmaster` process forks a fault probe process, `ftsprobe`. This is also known as the FTS (Fault Tolerance Server) process. The `postmaster` process restarts the FTS if it fails.

The FTS runs in a loop with a sleep interval between each cycle. On each loop, the FTS probes each primary segment instance by making a TCP socket connection to the segment instance using the hostname and port registered in the `gp_segment_configuration` table. If the connection succeeds, the segment performs a few simple checks and reports back to the FTS. The checks include executing a `stat` system call on critical segment directories and checking for internal faults in the segment instance. If no issues are detected, a positive reply is sent to the FTS and no action is taken for that segment instance.

If the connection cannot be made, or if a reply is not received in the timeout period, then a retry is attempted for the segment instance. If the configured maximum number of probe attempts fail, the FTS probes the segment's mirror to ensure that it is up, and then updates the `gp_segment_configuration`

table, marking the primary segment "down" and setting the mirror to act as the primary. The FTS updates the *gp_configuration_history* table with the operations performed.

When there is only an active primary segment and the corresponding mirror is down, the primary goes into the *not synchronizing* state and continues logging database changes, so the mirror can be synchronized without performing a full copy of data from the primary to the mirror.

Running the `gpstate` utility with the `-e` option displays any issues with a primary or mirror segment instances. Other `gpstate` options that display information about all primary or mirror segment instances such as `-m` (mirror instance information) and `-c` (primary and mirror configuration information) also display information about primary and mirror issues.

You can also can see the mode: `s` (synchronizing) or `n` (not synchronizing) for each segment instance, as well as the status `u` (up) or `d` (down), in the `gp_segment_configuration` table.

The *gprecoverseg* utility is used to bring up a mirror that is down. By default, `gprecoverseg` performs an incremental recovery, placing the mirror into synchronizing mode, which starts to replay the recorded changes from the primary onto the mirror. If the incremental recovery cannot be completed, the recovery fails and `gprecoverseg` should be run again with the `-F` option, to perform full recovery. This causes the primary to copy all of the data to the mirror.

After a segment instance has been recovered, the `gpstate -e` command might list primary and mirror segment instances that are switched. This indicates that the system is not balanced (the primary and mirror instances are not in their originally configured roles). If a system is not balanced, there might be skew resulting from the number of active primary segment instances on segment host systems.

The `gp_segment_configuration` table has columns `role` and `preferred_role`. These can have values of either `p` for primary or `m` for mirror. The `role` column shows the segment instance current role and the `preferred_role` shows the original role of the segment instance.

In a balanced system, the `role` and `preferred_role` matches for all segment instances. When they do not match the system is not balanced. To rebalance the cluster and bring all the segments into their preferred role, run the `gprecoverseg` command with the `-r` option.

Simple Failover and Recovery Example

Consider a single primary-mirror segment instance pair where the primary segment has failed over to the mirror. The following table shows the segment instance preferred role, role, mode, and status from `gp_segment_configuration` table before beginning recovery of the failed primary segment.

You can also run `gpstate -e` to display any issues with a primary or mirror segment instances.

	preferred_role	**role**	**mode**	**status**
Primary	p (primary)	m (mirror)	n (not synchronizing)	d (down)
Mirror	m (mirror)	p (primary)	n (not synchronizing)	u (up)

The segment instance roles are not in their preferred roles, and the primary is down. The mirror is up, the role is now primary, and it is not synchronizing because its mirror, the failed primary, is down. After fixing issues with the segment host and primary segment instance, you use `gprecoverseg` to prepare failed segment instances for recovery and initiate synchronization between the primary and mirror instances.

Once `gprecoverseg` has completed, the segments are in the states shown in the following table where the primary-mirror segment pair is up with the primary and mirror roles reversed from their preferred roles.

	preferred_role	role	mode	status
Primary	p (primary)	m (mirror)	s (synchronizing)	u (up)
Mirror	m (mirror)	p (primary)	s (synchronizing)	u (up)

The `gprecoverseg -r` command rebalances the system by returning the segment roles to their preferred roles.

	preferred_role	role	mode	status
Primary	p (primary)	p (primary)	s (synchronized)	u (up)
Mirror	m (mirror)	m (mirror)	s (synchronized)	u (up)

Configuring FTS Behavior

There is a set of server configuration parameters that affect FTS behavior:

`gp_fts_probe_interval`

How often, in seconds, to begin a new FTS loop. For example if the setting is 60 and the probe loop takes 10 seconds, the FTS process sleeps 50 seconds. If the setting is 60 and probe loop takes 75 seconds, the process sleeps 0 seconds. The default is 60, and the maximum is 3600.

`gp_fts_probe_timeout`

Probe timeout between master and segment, in seconds. The default is 20, and the maximum is 3600.

`gp_fts_probe_retries`

The number of attempts to probe a segment. For example if the setting is 5 there will be 4 retries after the first attempt fails. Default: 5

`gp_log_fts`

Logging level for FTS. The value may be "off", "terse", "verbose", or "debug". The "verbose" setting can be used in production to provide useful data for troubleshooting. The "debug" setting should not be used in production. Default: "terse"

`gp_segment_connect_timeout`

The maximum time (in seconds) allowed for a mirror to respond. Default: 600 (10 minutes)

In addition to the fault checking performed by the FTS, a primary segment that is unable to send data to its mirror can change the status of the mirror to down. The primary queues up the data and after `gp_segment_connect_timeout` seconds passes, indicates a mirror failure, causing the mirror to be marked down and the primary to go into change tracking mode.

Checking for Failed Segments

With mirroring enabled, you can have failed segment instances in the system without interruption of service or any indication that a failure has occurred. You can verify the status of your system using the `gpstate` utility. `gpstate` provides the status of each individual component of a Greenplum Database system, including primary segments, mirror segments, master, and standby master.

To check for failed segments

1. On the master host, run the *gpstate* utility with the `-e` option to show segment instances with error conditions:

   ```
   $ gpstate -e
   ```

   If the utility lists `Segments with Primary and Mirror Roles Switched`, the segment is not in its *preferred role* (the role to which it was assigned at system initialization). This means the system is in a potentially unbalanced state, as some segment hosts may have more active segments than is optimal for top system performance.

   Segments that display the `Config status` as `Down` indicate the corresponding mirror segment is down.

   See *Recovering From Segment Failures* for instructions to fix this situation.
2. To get detailed information about failed segments, you can check the *gp_segment_configuration* catalog table. For example:

   ```
   $ psql postgres -c "SELECT * FROM gp_segment_configuration WHERE
    status='d';"
   ```
3. For failed segment instances, note the host, port, preferred role, and data directory. This information will help determine the host and segment instances to troubleshoot.
4. To show information about mirror segment instances, run:

   ```
   $ gpstate -m
   ```

Checking the Log Files for Failed Segments

Log files can provide information to help determine an error's cause. The master and segment instances each have their own log file in `pg_log` of the data directory. The master log file contains the most information and you should always check it first.

Use the *gplogfilter* utility to check the Greenplum Database log files for additional information. To check the segment log files, run `gplogfilter` on the segment hosts using *gpssh*.

To check the log files

1. Use `gplogfilter` to check the master log file for `WARNING`, `ERROR`, `FATAL` or `PANIC` log level messages:

   ```
   $ gplogfilter -t
   ```
2. Use `gpssh` to check for `WARNING`, `ERROR`, `FATAL`, or `PANIC` log level messages on each segment instance. For example:

   ```
   $ gpssh -f seg_hosts_file -e 'source
   /usr/local/greenplum-db/greenplum_path.sh ; gplogfilter -t
   /data1/primary/*/pg_log/gpdb*.log' > seglog.out
   ```

Recovering a Failed Segment

If the master cannot connect to a segment instance, it marks that segment as down in the Greenplum Database system catalog. The segment instance remains offline until an administrator takes steps to bring the segment back online. The process for recovering a failed segment instance or host depends on the failure cause and whether or not mirroring is enabled. A segment instance can be unavailable for many reasons:

- A segment host is unavailable; for example, due to network or hardware failures.

- A segment instance is not running; for example, there is no `postgres` database listener process.
- The data directory of the segment instance is corrupt or missing; for example, data is not accessible, the file system is corrupt, or there is a disk failure.

Figure 18: Segment Failure Troubleshooting Matrix shows the high-level steps for each of the preceding failure scenarios.

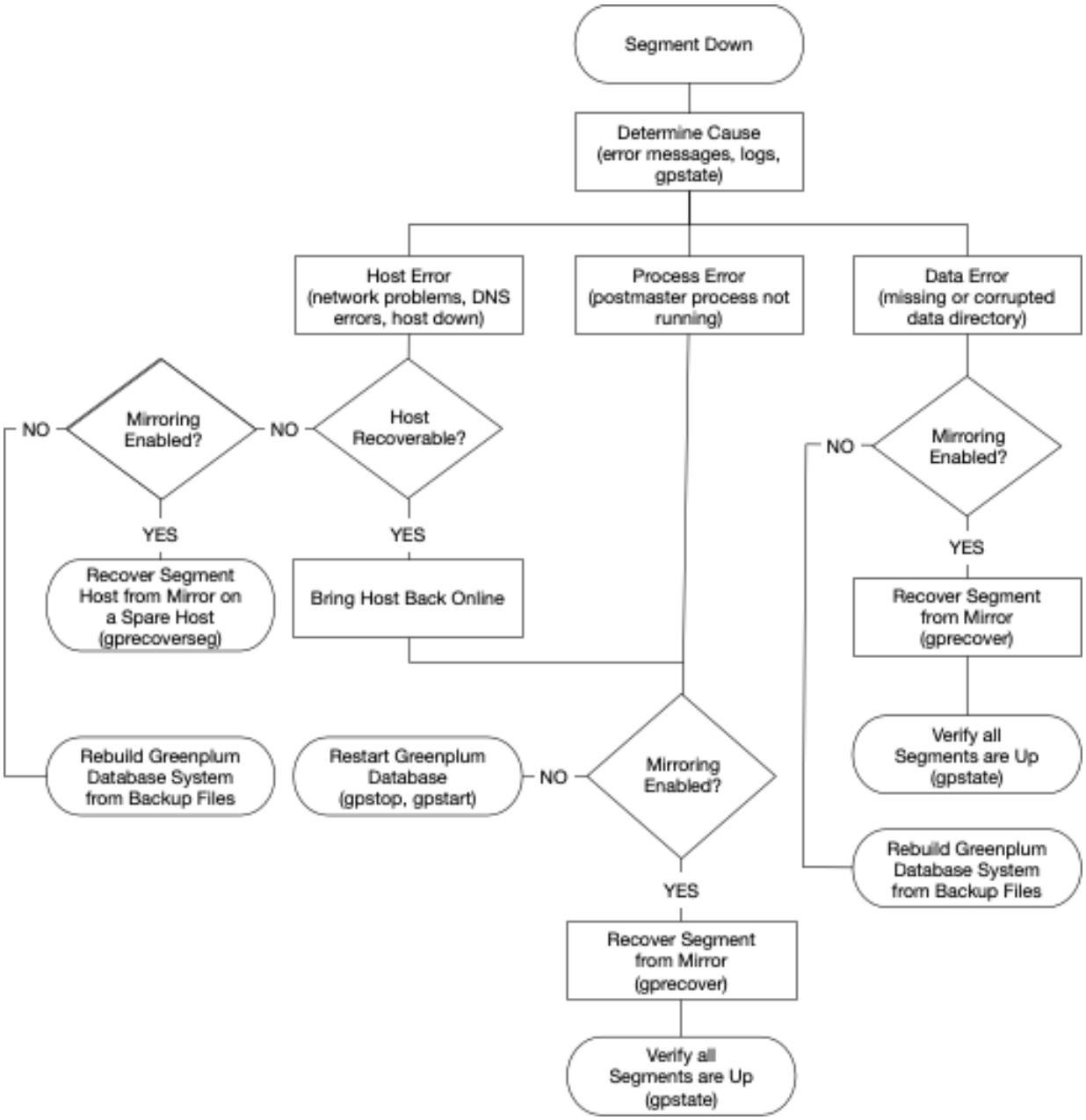


Figure 18: Segment Failure Troubleshooting Matrix

Recovering From Segment Failures

Segment host failures usually cause multiple segment failures: all primary or mirror segment instances on the host are marked as down and nonoperational. If mirroring is not enabled and a segment goes down, the system automatically becomes nonoperational.

A segment instance can fail for several reasons, such as a host failure, network failure, or disk failure. When a segment instance fails, its status is marked as *down* in the Greenplum Database system catalog, and its mirror is activated in *change tracking* mode. In order to bring the failed segment instance back into

operation again, you must first correct the problem that made it fail in the first place, and then recover the segment instance in Greenplum Database using `gprecoverseg`.

If a segment host is not recoverable and you have lost one or more segment instances with mirroring enabled, you can attempt to recover a segment instance from its mirror. See *When a segment host is not recoverable*. You can also recreate your Greenplum Database system from backup files. See *Backing Up and Restoring Databases*.

To recover with mirroring enabled

1. Ensure you can connect to the segment host from the master host. For example:

   ```
   $ ping failed_seg_host_address
   ```

2. Troubleshoot the problem that prevents the master host from connecting to the segment host. For example, the host machine may need to be restarted or replaced.
3. After the host is online and you can connect to it, run the *gprecoverseg* utility from the master host to reactivate the failed segment instances and start the process of sycnronizing the master and mirror instances. For example:

   ```
   $ gprecoverseg
   ```

4. The recovery process brings up the failed segments and identifies the changed files that need to be synchronized. The process can take some time; wait for the process to complete.
5. After `gprecoverseg` completes, the system goes into *Resynchronizing* mode and begins copying the changed files. This process runs in the background while the system is online and accepting database requests.
6. When the resynchronization process completes, the system state is *Synchronized*. Run the *gpstate* utility to verify the status of the resynchronization process:

   ```
   $ gpstate -m
   ```

   Note: If incremental recovery was not successful and the down segment instance data is not corrupted, contact Pivotal Support.

To return all segments to their preferred role

When a primary segment instance goes down, the mirror activates and becomes the primary segment. After running `gprecoverseg`, the currently active segment instance remains the primary and the failed segment becomes the mirror. The segment instances are not returned to the preferred role that they were given at system initialization time. This means that the system could be in a potentially unbalanced state if segment hosts have more active segments than is optimal for top system performance. To check for unbalanced segments and rebalance the system, run:

```
$ gpstate -e
```

All segments must be online and fully synchronized to rebalance the system. Database sessions remain connected during rebalancing, but queries in progress are canceled and rolled back.

1. Run `gpstate -m` to ensure all mirrors are *Synchronized*.

   ```
   $ gpstate -m
   ```

2. If any mirrors are in *Resynchronizing* mode, wait for them to complete.
3. Run `gprecoverseg` with the `-r` option to return the segments to their preferred roles.

   ```
   $ gprecoverseg -r
   ```

4. After rebalancing, run `gpstate -e` to confirm all segments are in their preferred roles.

```
$ gpstate -e
```

To recover from a double fault

In a double fault, both a primary segment and its mirror are down. This can occur if hardware failures on different segment hosts happen simultaneously. Greenplum Database is unavailable if a double fault occurs.

To recover from a double fault.

1. Troubleshoot the problem that caused the double fault and ensure that the segment hosts are operational and are accessible from the master host.
2. Restart Greenplum Database. The *gpstop* option `-r` stops and restarts the system.

```
$ gpstop -r
```

3. After the system restarts, run `gprecoverseg` to reactivate the failed segment instances.

```
$ gprecoverseg
```

   Note: If incremental recovery was not successful and the down segment instance data is not corrupted, contact Pivotal Support.

4. After `gprecoverseg` completes, use `gpstate` to check the status of your mirrors and ensure the segment instances have gone from *Resynchronizing* mode to *Synchronized* mode:

```
$ gpstate -m
```

5. If you still have segment instances in *change tracking* mode, you can run `gprecoverseg` with the `-F` option to perform a full segment recovery.

   Warning: A full recovery deletes the data directory of the down segment instance before copying the data from the active (current primary) segment instance. Before performing a full recovery, ensure that the segment failure did not cause data corruption and that any host segment disk issues have been fixed.

```
$ gprecoverseg -F
```

6. If needed, return segment instances to their preferred role. See *To return all segments to their preferred role*.

To recover without mirroring enabled

1. Ensure you can connect to the segment host from the master host. For example:

```
$ ping failed_seg_host_address
```

2. Troubleshoot the problem that is preventing the master host from connecting to the segment host. For example, the host machine may need to be restarted.
3. After the host is online, verify that you can connect to it and restart Greenplum Database. The `gpstop` option `-r` stops and restarts the system:

```
$ gpstop -r
```

4. Run the `gpstate` utility to verify that all segment instances are online:

```
$ gpstate
```

When a segment host is not recoverable

If a host is nonoperational, for example, due to hardware failure, recover the segments onto a spare set of hardware resources. If mirroring is enabled, you can recover a segment instance from its mirror onto an alternate host using the *gprecoverseg* utility. For example:

```
$ gprecoverseg -i recover_config_file
```

Where the format of the *recover_config_file* file is:

```
<failed_host>|<port>|<data_dir>[ <recovery_host>|<port>|<recovery_data_dir>]
```

For example, to recover to a different host than the failed host without additional tablespaces configured (besides the default *pg_system* tablespace):

```
sdw1-1|50001|/data1/mirror/gpseg16 sdw4-1|50001|/data1/recover1/gpseg16
```

For information about creating a segment instance recovery file, see *gprecoverseg*.

The new recovery segment host must be pre-installed with the Greenplum Database software and configured exactly as the existing segment hosts.

Recovering a Failed Master

If the primary master fails, the Greenplum Database system is not accessible and WAL replication stops. Use *gpactivatestandby* to activate the standby master. Upon activation of the standby master, Greenplum Database reconstructs the master host state at the time of the last successfully committed transaction.

These steps assume a standby master host is configured for the system. See *Enabling Master Mirroring*.

To activate the standby master

1. Run the `gpactivatestandby` utility from the standby master host you are activating. For example:

   ```
   $ gpactivatestandby -d /data/master/gpseg-1
   ```

   Where `-d` specifies the data directory of the master host you are activating.

   After you activate the standby, it becomes the *active* or *primary* master for your Greenplum Database array.
2. After the utility completes, run `gpstate` with the `-b` option to display a summary of the system status:

   ```
   $ gpstate -b
   ```

   The master instance status should be `Active`. When a standby master is not configured, the command displays `No master standby configured` for the standby master status. If you configured a new standby master, its status is `Passive`.
3. Optional: If you have not already done so while activating the prior standby master, you can run `gpinitstandby` on the active master host to configure a new standby master.

   Important: You must initialize a new standby master to continue providing master mirroring.

   For information about restoring the original master and standby master configuration, see *Restoring Master Mirroring After a Recovery*.

Restoring Master Mirroring After a Recovery

After you activate a standby master for recovery, the standby master becomes the primary master. You can continue running that instance as the primary master if it has the same capabilities and dependability as the original master host.

You must initialize a new standby master to continue providing master mirroring unless you have already done so while activating the prior standby master. Run *gpinitstandby* on the active master host to configure a new standby master. See *Enabling Master Mirroring*.

You can restore the primary and standby master instances on the original hosts. This process swaps the roles of the primary and standby master hosts, and it should be performed only if you strongly prefer to run the master instances on the same hosts they occupied prior to the recovery scenario.

> **Important:** Restoring the primary and standby master instances to their original hosts is not an online operation. The master host must be stopped to perform the operation.

For information about the Greenplum Database utilities, see the *Greenplum Database Utility Guide*.

To restore the master and standby instances on original hosts (optional)

1. Ensure the original master host is in dependable running condition; ensure the cause of the original failure is fixed.
2. On the original master host, move or remove the data directory, `gpseg-1`. This example moves the directory to `backup_gpseg-1`:

   ```
   $ mv /data/master/gpseg-1 /data/master/backup_gpseg-1
   ```

   You can remove the backup directory once the standby is successfully configured.
3. Initialize a standby master on the original master host. For example, run this command from the current master host, smdw:

   ```
   $ gpinitstandby -s mdw
   ```
4. After the initialization completes, check the status of standby master, mdw. Run *gpstate* with the `-f` option to check the standby master status:

   ```
   $ gpstate -f
   ```

   The standby master status should be `passive`, and the WAL sender state should be `streaming`.
5. Stop the Greenplum Database master instance on the standby master. For example:

   ```
   $ gpstop -m
   ```
6. Run the `gpactivatestandby` utility from the original master host, mdw, that is currently a standby master. For example:

   ```
   $ gpactivatestandby -d $MASTER_DATA_DIRECTORY
   ```

   Where the `-d` option specifies the data directory of the host you are activating.
7. After the utility completes, run `gpstate` with the `-b` option to display a summary of the system status:

   ```
   $ gpstate -b
   ```

   The master instance status should be `Active`. When a standby master is not configured, the command displays `No master standby configured` for the standby master state.

8. On the standby master host, move or remove the data directory, `gpseg-1`. This example moves the directory:

   ```
   $ mv /data/master/gpseg-1 /data/master/backup_gpseg-1
   ```

   You can remove the backup directory once the standby is successfully configured.
9. After the original master host runs the primary Greenplum Database master, you can initialize a standby master on the original standby master host. For example:

   ```
   $ gpinitstandby -s smdw
   ```

   After the command completes, you can run the `gpstate -f` command on the primary master host, to check the standby master status.

To check the status of the master mirroring process (optional)

You can run the `gpstate` utility with the `-f` option to display details of the standby master host.

```
$ gpstate -f
```

The standby master status should be `passive`, and the WAL sender state should be `streaming`.

For information about the *`gpstate`* utility, see the *Greenplum Database Utility Guide*.

Backing Up and Restoring Databases

This topic describes how to use Greenplum backup and restore features.

Performing backups regularly ensures that you can restore your data or rebuild your Greenplum Database system if data corruption or system failure occur. You can also use backups to migrate data from one Greenplum Database system to another.

Backup and Restore Overview

Greenplum Database supports parallel and non-parallel methods for backing up and restoring databases. Parallel operations scale regardless of the number of segments in your system, because segment hosts each write their data to local disk storage simultaneously. With non-parallel backup and restore operations, the data must be sent over the network from the segments to the master, which writes all of the data to its storage. In addition to restricting I/O to one host, non-parallel backup requires that the master have sufficient local disk storage to store the entire database.

Parallel Backup with gpbackup and gprestore

`gpbackup` and `gprestore` are the Greenplum Database backup and restore utilities. `gpbackup` utilizes `ACCESS SHARE` locks at the individual table level, instead of `EXCLUSIVE` locks on the `pg_class` catalog table. This enables you to execute DML statements during the backup, such as `CREATE`, `ALTER`, `DROP`, and `TRUNCATE` operations, as long as those operations do not target the current backup set.

Backup files created with `gpbackup` are designed to provide future capabilities for restoring individual database objects along with their dependencies, such as functions and required user-defined datatypes. See *Parallel Backup with gpbackup and gprestore* for more information.

Non-Parallel Backup with pg_dump

The PostgreSQL `pg_dump` and `pg_dumpall` non-parallel backup utilities can be used to create a single dump file on the master host that contains all data from all active segments.

The PostgreSQL non-parallel utilities should be used only for special cases. They are much slower than using the Greenplum backup utilities since all of the data must pass through the master. Additionally, it is

often the case that the master host has insufficient disk space to save a backup of an entire distributed Greenplum database.

The `pg_restore` utility requires compressed dump files created by `pg_dump` or `pg_dumpall`. Before starting the restore, you should modify the `CREATE TABLE` statements in the dump files to include the Greenplum `DISTRIBUTED` clause. If you do not include the `DISTRIBUTED` clause, Greenplum Database assigns default values, which may not be optimal. For details, see `CREATE TABLE` in the *Greenplum Database Reference Guide*.

To perform a non-parallel restore using parallel backup files, you can copy the backup files from each segment host to the master host, and then load them through the master.

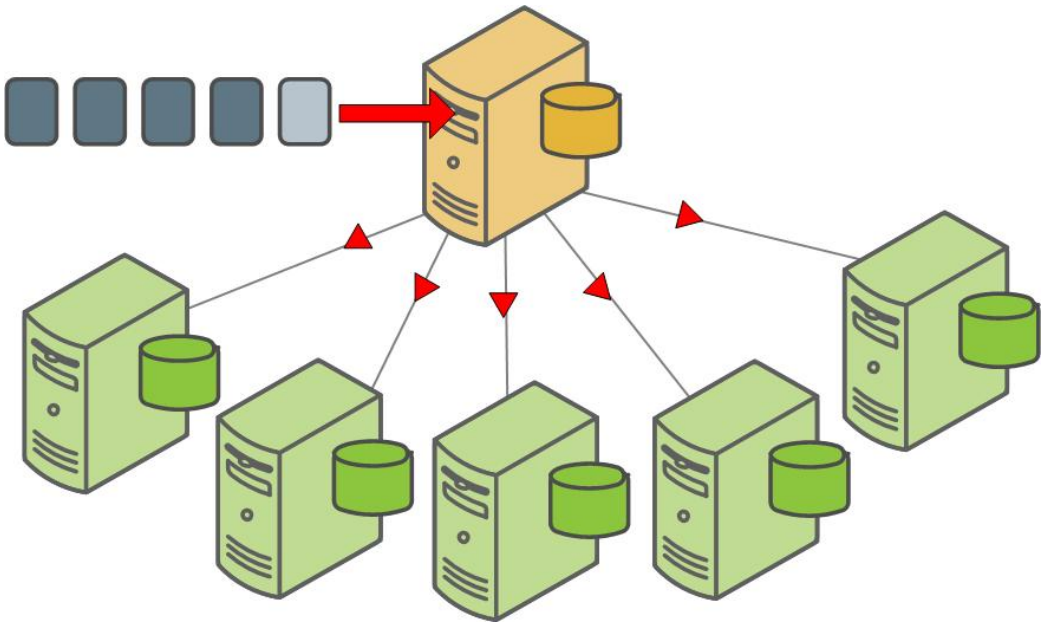


Figure 19: Non-parallel Restore Using Parallel Backup Files

Another non-parallel method for backing up Greenplum Database data is to use the `COPY TO` SQL command to copy all or a portion of a table out of the database to a delimited text file on the master host.

Parallel Backup with gpbackup and gprestore

`gpbackup` and `gprestore` are Greenplum Database utilities that create and restore backup sets for Greenplum Database. By default, `gpbackup` stores only the object metadata files and DDL files for a backup in the Greenplum Database master data directory. Greenplum Database segments use the `COPY ... ON SEGMENT` command to store their data for backed-up tables in compressed CSV data files, located in each segment's `backups` directory.

The backup metadata files contain all of the information that `gprestore` needs to restore a full backup set in parallel. Backup metadata also provides the framework for restoring only individual objects in the data set, along with any dependent objects, in future versions of `gprestore`. (See *Understanding Backup Files* for more information.) Storing the table data in CSV files also provides opportunities for using other restore utilities, such as `gpload`, to load the data either in the same cluster or another cluster. By default, one file is created for each table on the segment. You can specify the `--leaf-partition-data` option with `gpbackup` to create one data file per leaf partition of a partitioned table, instead of a single file. This option also enables you to filter backup sets by leaf partitions.

Each `gpbackup` task uses a single transaction in Greenplum Database. During this transaction, metadata is backed up on the master host, and data for each table on each segment host is written to CSV backup files using `COPY ... ON SEGMENT` commands in parallel. The backup process acquires an `ACCESS SHARE` lock on each table that is backed up.

For information about the gpbackup and gprestore utility options, see *gpbackup* and *gprestore*.

Requirements and Limitations

The gpbackup and gprestore utilities are compatible with these Greenplum Database versions:

- Pivotal Greenplum Database 4.3.22 and later
- Pivotal Greenplum Database 5.5.0 and later
- Pivotal Greenplum Database 6.0.0 and later

gpbackup and gprestore have the following limitations:

- If you create an index on a parent partitioned table, gpbackup does not back up that same index on child partitioned tables of the parent, as creating the same index on a child would cause an error. However, if you exchange a partition, gpbackup does not detect that the index on the exchanged partition is inherited from the new parent table. In this case, gpbackup backs up conflicting CREATE INDEX statements, which causes an error when you restore the backup set.
- You can execute multiple instances of gpbackup, but each execution requires a distinct timestamp.
- Database object filtering is currently limited to schemas and tables.
- When backing up a partitioned table where some or all leaf partitions are in different schemas from the root partition, the leaf partition table definitions, including the schemas, are backed up as metadata. This occurs even if the backup operation specifies that schemas that contain the leaf partitions should be excluded. To control data being backed up for this type of partitioned table in this situation, use the --leaf-partition-data option.
  - If the --leaf-partition-data option is not specified, the leaf partition data is also backed up even if the backup operation specifies that the leaf partition schemas should excluded.
  - If the --leaf-partition-data option is specified, the leaf partition data is not be backed up if the backup operation specifies that the leaf partition schemas should excluded. Only the metadata for leaf partition tables are backed up.
- If you use the gpbackup --single-data-file option to combine table backups into a single file per segment, you cannot perform a parallel restore operation with gprestore (cannot set --jobs to a value higher than 1).
- You cannot use the --exclude-table-file with --leaf-partition-data. Although you can specify leaf partition names in a file specified with --exclude-table-file, gpbackup ignores the partition names.
- Backing up a database with gpbackup while simultaneously running DDL commands might cause gpbackup to fail, in order to ensure consistency within the backup set. For example, if a table is dropped after the start of the backup operation, gpbackup exits and displays the error message ERROR: relation <*schema.table*> does not exist.

  gpbackup might fail when a table is dropped during a backup operation due to table locking issues. gpbackup generates a list of tables to back up and acquires an ACCESS SHARED lock on the tables. If an EXCLUSIVE LOCK is held on a table, gpbackup acquires the ACCESS SHARED lock after the existing lock is released. If the table no longer exists when gpbackup attempts to acquire a lock on the table, gpbackup exits with the error message.

  For tables that might be dropped during a backup, you can exclude the tables from a backup with a gpbackup table filtering option such as --exclude-table or --exclude-schema.
- A backup created with gpbackup can only be restored to a Greenplum Database cluster with the same number of segment instances as the source cluster. If you run gpexpand to add segments to the cluster, backups you made before starting the expand cannot be restored after the expansion has completed.

Objects Included in a Backup or Restore

The following table lists the objects that are backed up and restored with gpbackup and gprestore. Database objects are backed up for the database you specify with the --dbname option. Global objects

(Greenplum Database system objects) are also backed up by default, but they are restored only if you include the `--with-globals` option to `gprestore`.

Table 21: Objects that are backed up and restored

Database (for database specified with `--dbname`)	Global (requires the `--with-globals` option to restore)
• Session-level configuration parameter settings (GUCs) • Schemas, see *Note* • Procedural language extensions • Sequences • Comments • Tables • Indexes • Owners • Writable External Tables (DDL only) • Readable External Tables (DDL only) • Functions • Aggregates • Casts • Types • Views • Materialized Views (DDL only) • Protocols • Triggers. (While Greenplum Database does not support triggers, any trigger definitions that are present are backed up and restored.) • Rules • Domains • Operators, operator families, and operator classes • Conversions • Extensions • Text search parsers, dictionaries, templates, and configurations	• Tablespaces • Databases • Database-wide configuration parameter settings (GUCs) • Resource group definitions • Resource queue definitions • Roles • `GRANT` assignments of roles to databases

Note: These schemas are not included in a backup.

- `gp_toolkit`
- `information_schema`
- `pg_aoseg`
- `pg_bitmapindex`
- `pg_catalog`
- `pg_toast*`
- `pg_temp*`

When restoring to an existing database, `gprestore` assumes the `public` schema exists when restoring objects to the `public` schema. When restoring to a new database (with the `--create-db` option), `gprestore` creates the `public` schema automatically when creating a database with the `CREATE DATABASE` command. The command uses the `template0` database that contains the `public` schema.

See also *Understanding Backup Files*.

Performing Basic Backup and Restore Operations

To perform a complete backup of a database, as well as Greenplum Database system metadata, use the command:

```
$ gpbackup --dbname <database_name>
```

For example:

```
$ gpbackup --dbname demo
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-
Starting backup of database demo
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-Backup
 Timestamp = 20180105112754
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-Backup
 Database = demo
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-Backup
 Type = Unfiltered Compressed Full Backup
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-
Gathering list of tables for backup
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-
Acquiring ACCESS SHARE locks on tables
Locks acquired:  6 / 6
 [================================================================] 100.00%
 0s
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-
Gathering additional table metadata
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-
Writing global database metadata
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-Global
 database metadata backup complete
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-
Writing pre-data metadata
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-Pre-
data metadata backup complete
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-
Writing post-data metadata
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-Post-
data metadata backup complete
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-
Writing data to file
Tables backed up:  3 / 3
 [================================================================] 100.00% 0s
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-Data
 backup complete
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-Found
 neither /usr/local/greenplum-db/./bin/gp_email_contacts.yaml nor /home/
gpadmin/gp_email_contacts.yaml
20180105:11:27:54 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-Email
 containing gpbackup report /gpmaster/seg-1/backups/20180105/20180105112754/
gpbackup_20180105112754_report will not be sent
20180105:11:27:55 gpbackup:gpadmin:centos6.localdomain:002182-[INFO]:-Backup
 completed successfully
```

The above command creates a file that contains global and database-specific metadata on the Greenplum Database master host in the default directory, `$MASTER_DATA_DIRECTORY/backups/<YYYYMMDD>/<YYYYMMDDHHMMSS>/`. For example:

```
$ ls /gpmaster/gpsne-1/backups/20180105/20180105112754
gpbackup_20180105112754_config.yaml   gpbackup_20180105112754_report
```

```
gpbackup_20180105112754_metadata.sql  gpbackup_20180105112754_toc.yaml
```

By default, each segment stores each table's data for the backup in a separate compressed CSV file in `<seg_dir>/backups/<YYYYMMDD>/<YYYYMMDDHHMMSS>/`:

```
$ ls /gpdata1/gpsne0/backups/20180105/20180105112754/
gpbackup_0_20180105112754_17166.gz  gpbackup_0_20180105112754_26303.gz
gpbackup_0_20180105112754_21816.gz
```

To consolidate all backup files into a single directory, include the `--backup-dir` option. Note that you must specify an absolute path with this option:

```
$ gpbackup --dbname demo --backup-dir /home/gpadmin/backups
20171103:15:31:56 gpbackup:gpadmin:0ee2f5fb02c9:017586-[INFO]:-Starting
 backup of database demo
...
20171103:15:31:58 gpbackup:gpadmin:0ee2f5fb02c9:017586-[INFO]:-Backup
 completed successfully
$ find /home/gpadmin/backups/ -type f
/home/gpadmin/backups/gpseg0/backups/20171103/20171103153156/
gpbackup_0_20171103153156_16543.gz
/home/gpadmin/backups/gpseg0/backups/20171103/20171103153156/
gpbackup_0_20171103153156_16524.gz
/home/gpadmin/backups/gpseg1/backups/20171103/20171103153156/
gpbackup_1_20171103153156_16543.gz
/home/gpadmin/backups/gpseg1/backups/20171103/20171103153156/
gpbackup_1_20171103153156_16524.gz
/home/gpadmin/backups/gpseg-1/backups/20171103/20171103153156/
gpbackup_20171103153156_config.yaml
/home/gpadmin/backups/gpseg-1/backups/20171103/20171103153156/
gpbackup_20171103153156_predata.sql
/home/gpadmin/backups/gpseg-1/backups/20171103/20171103153156/
gpbackup_20171103153156_global.sql
/home/gpadmin/backups/gpseg-1/backups/20171103/20171103153156/
gpbackup_20171103153156_postdata.sql
/home/gpadmin/backups/gpseg-1/backups/20171103/20171103153156/
gpbackup_20171103153156_report
/home/gpadmin/backups/gpseg-1/backups/20171103/20171103153156/
gpbackup_20171103153156_toc.yaml
```

When performing a backup operation, you can use the `--single-data-file` in situations where the additional overhead of multiple files might be prohibitive. For example, if you use a third party storage solution such as Data Domain with back ups.

> **Note:** Backing up a materialized view does not back up the materialized view data. Only the materialized view definition is backed up.

Restoring from Backup

To use `gprestore` to restore from a backup set, you must use the `--timestamp` option to specify the exact timestamp value (YYYYMMDDHHMMSS) to restore. Include the `--create-db` option if the database does not exist in the cluster. For example:

```
$ dropdb demo
$ gprestore --timestamp 20171103152558 --create-db
20171103:15:45:30 gprestore:gpadmin:0ee2f5fb02c9:017714-[INFO]:-Restore Key
 = 20171103152558
20171103:15:45:31 gprestore:gpadmin:0ee2f5fb02c9:017714-[INFO]:-Creating
 database
20171103:15:45:44 gprestore:gpadmin:0ee2f5fb02c9:017714-[INFO]:-Database
 creation complete
```

```
20171103:15:45:44 gprestore:gpadmin:0ee2f5fb02c9:017714-[INFO]:-Restoring
 pre-data metadata from /gpmaster/gpsne-1/backups/20171103/20171103152558/
gpbackup_20171103152558_predata.sql
20171103:15:45:45 gprestore:gpadmin:0ee2f5fb02c9:017714-[INFO]:-Pre-data
 metadata restore complete
20171103:15:45:45 gprestore:gpadmin:0ee2f5fb02c9:017714-[INFO]:-Restoring
 data
20171103:15:45:45 gprestore:gpadmin:0ee2f5fb02c9:017714-[INFO]:-Data restore
 complete
20171103:15:45:45 gprestore:gpadmin:0ee2f5fb02c9:017714-[INFO]:-Restoring
 post-data metadata from /gpmaster/gpsne-1/backups/20171103/20171103152558/
gpbackup_20171103152558_postdata.sql
20171103:15:45:45 gprestore:gpadmin:0ee2f5fb02c9:017714-[INFO]:-Post-data
 metadata restore complete
```

If you specified a custom `--backup-dir` to consolidate the backup files, include the same `--backup-dir` option when using `gprestore` to locate the backup files:

```
$ dropdb demo
$ gprestore --backup-dir /home/gpadmin/backups/ --timestamp 20171103153156
 --create-db
20171103:15:51:02 gprestore:gpadmin:0ee2f5fb02c9:017819-[INFO]:-Restore Key
 = 20171103153156
...
20171103:15:51:17 gprestore:gpadmin:0ee2f5fb02c9:017819-[INFO]:-Post-data
 metadata restore complete
```

`gprestore` does not attempt to restore global metadata for the Greenplum System by default. If this is required, include the `--with-globals` argument.

By default, `gprestore` uses 1 connection to restore table data and metadata. If you have a large backup set, you can improve performance of the restore by increasing the number of parallel connections with the `--jobs` option. For example:

```
$ gprestore --backup-dir /home/gpadmin/backups/ --timestamp 20171103153156
 --create-db --jobs 8
```

Test the number of parallel connections with your backup set to determine the ideal number for fast data recovery.

> **Note:** You cannot perform a parallel restore operation with `gprestore` if the backup combined table backups into a single file per segment with the `gpbackup` option `--single-data-file`.
>
> Restoring a materialized view does not restore materialized view data. Only the materialized view definition is restored. To populate the materialized view with data, use `REFRESH MATERIALIZED VIEW`. The tables that are referenced by the materialized view definition must be available when you refresh the materialized view. The `gprestore` log file lists the materialized views that were restored and the `REFRESH MATERIALIZED VIEW` commands that are used to populate the materialized views with data.

Report Files

When performing a backup or restore operation, `gpbackup` and `gprestore` generate a report file. When email notification is configured, the email sent contains the contents of the report file. For information about email notification, see *Configuring Email Notifications*.

The report file is placed in the Greenplum Database master backup directory. The report file name contains the timestamp of the operation. These are the formats of the `gpbackup` and `gprestore` report file names.

```
gpbackup_<backup_timestamp>_report
gprestore_<backup_timestamp>_<restore_timesamp>_report
```

For these example report file names, `20180213114446` is the timestamp of the backup and `20180213115426` is the timestamp of the restore operation.

```
gpbackup_20180213114446_report
gprestore_20180213114446_20180213115426_report
```

This backup directory on a Greenplum Database master host contains both a `gpbackup` and `gprestore` report file.

```
$ ls -l /gpmaster/seg-1/backups/20180213/20180213114446
total 36
-r--r--r--. 1 gpadmin gpadmin  295 Feb 13 11:44
 gpbackup_20180213114446_config.yaml
-r--r--r--. 1 gpadmin gpadmin 1855 Feb 13 11:44
 gpbackup_20180213114446_metadata.sql
-r--r--r--. 1 gpadmin gpadmin 1402 Feb 13 11:44
 gpbackup_20180213114446_report
-r--r--r--. 1 gpadmin gpadmin 2199 Feb 13 11:44
 gpbackup_20180213114446_toc.yaml
-r--r--r--. 1 gpadmin gpadmin  404 Feb 13 11:54
 gprestore_20180213114446_20180213115426_report
```

The contents of the report files are similar. This is an example of the contents of a `gprestore` report file.

```
Greenplum Database Restore Report

Timestamp Key: 20180213114446
GPDB Version: 5.4.1+dev.8.g9f83645 build
 commit:9f836456b00f855959d52749d5790ed1c6efc042
gprestore Version: 1.0.0-alpha.3+dev.73.g0406681

Database Name: test
Command Line: gprestore --timestamp 20180213114446 --with-globals --createdb

Start Time: 2018-02-13 11:54:26
End Time: 2018-02-13 11:54:31
Duration: 0:00:05

Restore Status: Success
```

History File

When performing a backup operation, `gpbackup` appends backup information in the gpbackup history file, `gpbackup_history.yaml`, in the Greenplum Database master data directory. The file contains the backup timestamp, information about the backup options, and backup set information for incremental backups. This file is not backed up by `gpbackup`.

`gpbackup` uses the information in the file to find a matching backup for an incremental backup when you run `gpbackup` with the `--incremental` option and do not specify the `--from-timesamp` option to indicate the backup that you want to use as the latest backup in the incremental backup set. For information about incremental backups, see *Creating and Using Incremental Backups with gpbackup and gprestore*.

Return Codes

One of these codes is returned after `gpbackup` or `gprestore` completes.

- **0** – Backup or restore completed with no problems
- **1** – Backup or restore completed with non-fatal errors. See log file for more information.
- **2** – Backup or restore failed with a fatal error. See log file for more information.

Filtering the Contents of a Backup or Restore

`gpbackup` backs up all schemas and tables in the specified database, unless you exclude or include individual schema or table objects with schema level or table level filter options.

The schema level options are `--include-schema`, `--include-schema-file`, or `--exclude-schema`, `--exclude-schema-file` command-line options to `gpbackup`. For example, if the "demo" database includes only two schemas, "wikipedia" and "twitter," both of the following commands back up only the "wikipedia" schema:

```
$ gpbackup --dbname demo --include-schema wikipedia
$ gpbackup --dbname demo --exclude-schema twitter
```

You can include multiple `--include-schema` options in a `gpbackup` *or* multiple `--exclude-schema` options. For example:

```
$ gpbackup --dbname demo --include-schema wikipedia --include-schema twitter
```

If you have a large number of schemas, you can list the schemas in a text file and specify the file with the `--include-schema-file` or `--exclude-schema-file` options in a `gpbackup` command. Each line in the file must define a single schema, and the file cannot contain trailing lines. For example, this command uses a file in the `gpadmin` home directory to include a set of schemas.

```
gpbackup --dbname demo --include-schema-file /users/home/gpadmin/backup-
schemas
```

To filter the individual tables that are included in a backup set, or excluded from a backup set, specify individual tables with the `--include-table` option or the `--exclude-table` option. The table must be schema qualified, `<schema-name>.<table-name>`. The individual table filtering options can be specified multiple times. However, `--include-table` and `--exclude-table` cannot both be used in the same command.

You can create a list of qualified table names in a text file. When listing tables in a file, each line in the text file must define a single table using the format `<schema-name>.<table-name>`. The file must not include trailing lines. For example:

```
wikipedia.articles
twitter.message
```

If a table or schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. For example:

```
beer."IPA"
"Wine".riesling
"Wine"."sauvignon blanc"
water.tonic
```

After creating the file, you can use it either to include or exclude tables with the `gpbackup` options `--include-table-file` or `--exclude-table-file`. For example:

```
$ gpbackup --dbname demo --include-table-file /home/gpadmin/table-list.txt
```

You can combine `-include schema` with `--exclude-table` or `--exclude-table-file` for a backup. This example uses `--include-schema` with `--exclude-table` to back up a schema except for a single table.

```
$ gpbackup --dbname demo --include-schema mydata --exclude-table
 mydata.addresses
```

You cannot combine `--include-schema` with `--include-table` or `--include-table-file`, and you cannot combine `--exclude-schema` with any table filtering option such as `--exclude-table` or `--include-table`.

When you use `--include-table` or `--include-table-file` dependent objects are not automatically backed up or restored, you must explicitly specify the dependent objects that are required. For example, if you back up or restore a view or materialized view, you must also specify the tables that the view or the materialized view uses. If you backup or restore a table that uses a sequence, you must also specify the sequence.

Filtering by Leaf Partition

By default, `gpbackup` creates one file for each table on a segment. You can specify the `--leaf-partition-data` option to create one data file per leaf partition of a partitioned table, instead of a single file. You can also filter backups to specific leaf partitions by listing the leaf partition names in a text file to include. For example, consider a table that was created using the statement:

```
demo=# CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( PARTITION Jan17 START (date '2017-01-01') INCLUSIVE ,
PARTITION Feb17 START (date '2017-02-01') INCLUSIVE ,
PARTITION Mar17 START (date '2017-03-01') INCLUSIVE ,
PARTITION Apr17 START (date '2017-04-01') INCLUSIVE ,
PARTITION May17 START (date '2017-05-01') INCLUSIVE ,
PARTITION Jun17 START (date '2017-06-01') INCLUSIVE ,
PARTITION Jul17 START (date '2017-07-01') INCLUSIVE ,
PARTITION Aug17 START (date '2017-08-01') INCLUSIVE ,
PARTITION Sep17 START (date '2017-09-01') INCLUSIVE ,
PARTITION Oct17 START (date '2017-10-01') INCLUSIVE ,
PARTITION Nov17 START (date '2017-11-01') INCLUSIVE ,
PARTITION Dec17 START (date '2017-12-01') INCLUSIVE
END (date '2018-01-01') EXCLUSIVE );
NOTICE:  CREATE TABLE will create partition "sales_1_prt_jan17" for table
 "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_feb17" for table
 "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_mar17" for table
 "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_apr17" for table
 "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_may17" for table
 "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_jun17" for table
 "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_jul17" for table
 "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_aug17" for table
 "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_sep17" for table
 "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_oct17" for table
 "sales"
```

```
NOTICE:  CREATE TABLE will create partition "sales_1_prt_nov17" for table
 "sales"
NOTICE:  CREATE TABLE will create partition "sales_1_prt_dec17" for table
 "sales"
CREATE TABLE
```

To back up only data for the last quarter of the year, first create a text file that lists those leaf partition names instead of the full table name:

```
public.sales_1_prt_oct17
public.sales_1_prt_nov17
public.sales_1_prt_dec17
```

Then specify the file with the `--include-table-file` option to generate one data file per leaf partition:

```
$ gpbackup --dbname demo --include-table-file last-quarter.txt --leaf-
partition-data
```

When you specify `--leaf-partition-data`, `gpbackup` generates one data file per leaf partition when backing up a partitioned table. For example, this command generates one data file for each leaf partition:

```
$ gpbackup --dbname demo --include-table public.sales --leaf-partition-data
```

When leaf partitions are backed up, the leaf partition data is backed up along with the metadata for the entire partitioned table.

> **Note:** You cannot use the `--exclude-table-file` option with `--leaf-partition-data`. Although you can specify leaf partition names in a file specified with `--exclude-table-file`, `gpbackup` ignores the partition names.

Filtering with gprestore

After creating a backup set with `gpbackup`, you can filter the schemas and tables that you want to restore from the backup set using the `gprestore` `--include-schema` and `--include-table-file` options. These options work in the same way as their `gpbackup` counterparts, but have the following restrictions:

- The tables that you attempt to restore must not already exist in the database.
- If you attempt to restore a schema or table that does not exist in the backup set, the `gprestore` does not execute.
- If you use the `--include-schema` option, `gprestore` cannot restore objects that have dependencies on multiple schemas.
- If you use the `--include-table-file` option, `gprestore` does not create roles or set the owner of the tables. The utility restores table indexes and rules. Triggers are also restored but are not supported in Greenplum Database.
- The file that you specify with `--include-table-file` cannot include a leaf partition name, as it can when you specify this option with `gpbackup`. If you specified leaf partitions in the backup set, specify the partitioned table to restore the leaf partition data.

  When restoring a backup set that contains data from some leaf partitions of a partitioned table, the partitioned table is restored along with the data for the leaf partitions. For example, you create a backup with the `gpbackup` option `--include-table-file` and the text file lists some leaf partitions of a partitioned table. Restoring the backup creates the partitioned table and restores the data only for the leaf partitions listed in the file.

Configuring Email Notifications

`gpbackup` and `gprestore` can send email notifications after a back up or restore operation completes.

To have gpbackup or gprestore send out status email notifications, you must place a file named gp_email_contacts.yaml in the home directory of the user running gpbackup or gprestore in the same directory as the utilities ($GPHOME/bin). A utility issues a message if it cannot locate a gp_email_contacts.yaml file in either location. If both locations contain a .yaml file, the utility uses the file in user $HOME.

The email subject line includes the utility name, timestamp, status, and the name of the Greenplum Database master. This is an example subject line for a gpbackup email.

```
gpbackup 20180202133601 on gp-master completed
```

The email contains summary information about the operation including options, duration, and number of objects backed up or restored. For information about the contents of a notification email, see *Report Files*.

> **Note:** The UNIX mail utility must be running on the Greenplum Database host and must be configured to allow the Greenplum superuser (gpadmin) to send email. Also ensure that the mail program executable is locatable via the gpadmin user's $PATH.

gpbackup and gprestore Email File Format

The gpbackup and gprestore email notification YAML file gp_email_contacts.yaml uses indentation (spaces) to determine the document hierarchy and the relationships of the sections to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.

> **Note:** If the status parameters are not specified correctly, the utility does not issue a warning. For example, if the success parameter is misspelled and is set to true, a warning is not issued and an email is not sent to the email address after a successful operation. To ensure email notification is configured correctly, run tests with email notifications configured.

This is the format of the gp_email_contacts.yaml YAML file for gpbackup email notifications:

```
contacts:
  gpbackup:
  - address: user@domain
    status:
         success: [true | false]
         success_with_errors: [true | false]
         failure: [true | false]
  gprestore:
  - address: user@domain
    status:
         success: [true | false]
         success_with_errors: [true | false]
         failure: [true | false]
```

Email YAML File Sections

contacts

Required. The section that contains the gpbackup and gprestore sections. The YAML file can contain a gpbackup section, a gprestore section, or one of each.

gpbackup

Optional. Begins the gpbackup email section.

address

Required. At least one email address must be specified. Multiple email address parameters can be specified. Each address requires a status section.

user@domain is a single, valid email address.

status

Required. Specify when the utility sends an email to the specified email address. The default is to not send email notification.

You specify sending email notifications based on the completion status of a backup or restore operation. At least one of these parameters must be specified and each parameter can appear at most once.

`success`

Optional. Specify if an email is sent if the operation completes without errors. If the value is `true`, an email is sent if the operation completes without errors. If the value is `false` (the default), an email is not sent.

`success_with_errors`

Optional. Specify if an email is sent if the operation completes with errors. If the value is `true`, an email is sent if the operation completes with errors. If the value is `false` (the default), an email is not sent.

`failure`

Optional. Specify if an email is sent if the operation fails. If the value is `true`, an email is sent if the operation fails. If the value is `false` (the default), an email is not sent.

`gprestore`

Optional. Begins the `gprestore` email section. This section contains the *`address`* and *`status`* parameters that are used to send an email notification after a `gprestore` operation. The syntax is the same as the *`gpbackup`* section.

Examples

This example YAML file specifies sending email to email addresses depending on the success or failure of an operation. For a backup operation, an email is sent to a different address depending on the success or failure of the backup operation. For a restore operation, an email is sent to `gpadmin@example.com` only when the operation succeeds or completes with errors.

```
contacts:
  gpbackup:
  - address: gpadmin@example.com
    status:
      success:true
  - address: my_dba@example.com
    status:
      success_with_errors: true
      failure: true
  gprestore:
  - address: gpadmin@example.com
    status:
      success: true
      success_with_errors: true
```

Understanding Backup Files

Warning: All `gpbackup` metadata files are created with read-only permissions. Never delete or modify the metadata files for a `gpbackup` backup set. Doing so will render the backup files non-functional.

A complete backup set for `gpbackup` includes multiple metadata files, supporting files, and CSV data files, each designated with the timestamp at which the backup was created.

By default, metadata and supporting files are stored on the Greenplum Database master host in the directory `$MASTER_DATA_DIRECTORY/backups/YYYYMMDD/YYYYMMDDHHMMSS/`. If you specify a custom backup directory, this same file path is created as a subdirectory of the backup directory. The following table describes the names and contents of the metadata and supporting files.

Table 22: gpbackup Metadata Files (master)

File name	Description
`gpbackup_<YYYYMMDDHHMMSS>_metadata.sql`	Contains global and database-specific metadata: • DDL for objects that are global to the Greenplum Database cluster, and not owned by a specific database within the cluster. • DDL for objects in the backed-up database (specified with `--dbname`) that must be created *before* to restoring the actual data, and DDL for objects that must be created *after* restoring the data. Global objects include: • Tablespaces • Databases • Database-wide configuration parameter settings (GUCs) • Resource group definitions • Resource queue definitions • Roles • `GRANT` assignments of roles to databases **Note:** Global metadata is not restored by default. You must include the `--with-globals` option to the `gprestore` command to restore global metadata. Database-specific objects that must be created *before* to restoring the actual data include: • Session-level configuration parameter settings (GUCs) • Schemas • Procedural language extensions • Types • Sequences • Functions • Tables • Protocols • Operators and operator classes • Conversions • Aggregates • Casts • Views • Materialized Views **Note:** Materialized view data is not restored, only the definition. • Constraints Database-specific objects that must be created *after* restoring the actual data include: • Indexes • Rules

File name	Description
	• Triggers. (While Greenplum Database does not support triggers, any trigger definitions that are present are backed up and restored.)
`gpbackup_<YYYYMMDDHHMMSS>_toc.yaml`	Contains metadata for locating object DDL in the `_predata.sql` and `_postdata.sql` files. This file also contains the table names and OIDs used for locating the corresponding table data in CSV data files that are created on each segment. See *Segment Data Files*.
`gpbackup_<YYYYMMDDHHMMSS>_report`	Contains information about the backup operation that is used to populate the email notice (if configured) that is sent after the backup completes. This file contains information such as: • Command-line options that were provided • Database that was backed up • Database version • Backup type See *Configuring Email Notifications*.
`gpbackup_<YYYYMMDDHHMMSS>_config.yaml`	Contains metadata about the execution of the particular backup task, including: • `gpbackup` version • Database name • Greenplum Database version • Additional option settings such as `--no-compression`, `--compression-level`, `--metadata-only`, `--data-only`, and `--with-stats`.
`gpbackup_history.yaml`	Contains information about options that were used when creating a backup with `gpbackup`, and information about incremental backups. Stored on the Greenplum Database master host in the Greenplum Database master data directory. This file is not backed up by `gpbackup`. For information about incremental backups, see *Creating and Using Incremental Backups with gpbackup and gprestore*.

Segment Data Files

By default, each segment creates one compressed CSV file for each table that is backed up on the segment. You can optionally specify the `--single-data-file` option to create a single data file on each segment. The files are stored in `<seg_dir>/backups/YYYYMMDD/YYYYMMDDHHMMSS/`.

If you specify a custom backup directory, segment data files are copied to this same file path as a subdirectory of the backup directory. If you include the `--leaf-partition-data` option, `gpbackup` creates one data file for each leaf partition of a partitioned table, instead of just one table for file.

Each data file uses the file name format `gpbackup_<content_id>_<YYYYMMDDHHMMSS>_<oid>.gz` where:

- <content_id> is the content ID of the segment.
- <YYYYMMDDHHMMSS> is the timestamp of the gpbackup operation.
- <oid> is the object ID of the table. The metadata file gpbackup_<YYYYMMDDHHMMSS>_toc.yaml references this <oid> to locate the data for a specific table in a schema.

You can optionally specify the gzip compression level (from 1-9) using the --compression-level option, or disable compression entirely with --no-compression. If you do not specify a compression level, gpbackup uses compression level 1 by default.

Creating and Using Incremental Backups with gpbackup and gprestore

The gpbackup and gprestore utilities support creating incremental backups of append-optimized tables and restoring from incremental backups. An incremental backup backs up all specified heap tables and backs up append-optimized tables (including append-optimized, column-oriented tables) only if the tables have changed. For example, if a row of an append-optimized table has changed, the table is backed up. For partitioned append-optimized tables, only the changed leaf partitions are backed up.

Incremental backups are efficient when the total amount of data in append-optimized tables or table partitions that changed is small compared to the data that has not changed since the last backup.

An incremental backup backs up an append-optimized table only if one of the following operations was performed on the table after the last full or incremental backup:

- ALTER TABLE
- DELETE
- INSERT
- TRUNCATE
- UPDATE
- DROP and then re-create the table

To restore data from incremental backups, you need a complete incremental backup set.

About Incremental Backup Sets

An incremental backup set includes the following backups:

- A full backup. This is the full backup that the incremental backups are based on.
- The set of incremental backups that capture the changes to the database from the time of the full backup.

For example, you can create a full backup and then create three daily incremental backups. The full backup and all three incremental backups are the backup set. For information about using an incremental backup set, see *Example Using Incremental Backup Sets*.

When you create or add to an incremental backup set, gpbackup ensures that the backups in the set are created with a consistent set of backup options to ensure that the backup set can be used in a restore operation. For information about backup set consistency, see *Using Incremental Backups*.

When you create an incremental backup you include these options with the other gpbackup options to create a backup:

- --leaf-partition-data - Required for all backups in the incremental backup set.
  - Required when you create a full backup that will be the base backup for an incremental backup set.
  - Required when you create an incremental backup.
- --incremental - Required when you create an incremental backup.

  You cannot combine --data-only or --metadata-only with --incremental.
- --from-timestamp - Optional. This option can be used with --incremental. The timestamp you specify is an existing backup. The timestamp can be either a full backup or incremental backup. The

backup being created must be compatible with the backup specified with the `--from-timestamp` option.

If you do not specify `--from-timestamp`, `gpbackup` attempts to find a compatible backup based on information in the `gpbackup` history file. See *Incremental Backup Notes*.

Using Incremental Backups

- *Example Using Incremental Backup Sets*
- *Creating an Incremental Backup with gpbackup*
- *Restoring from an Incremental Backup with gprestore*
- *Incremental Backup Notes*

When you add an incremental backup to a backup set, `gpbackup` ensures that the full backup and the incremental backups are consistent by checking these `gpbackup` options:

- `--dbname` - The database must be the same.
- `--backup-dir` - The directory must be the same. The backup set, the full backup and the incremental backups, must be in the same location.
- `--single-data-file` - This option must be either specified or absent for all backups in the set.
- `--plugin-config` - If this option is specified, it must be specified for all backups in the backup set. The configuration must reference the same plugin binary.
- `--include-table-file`, `--include-schema`, or any other options that filter tables and schemas must be the same.

  When checking schema filters, only the schema names are checked, not the objects contained in the schemas.
- `--no-compression` - If this option is specified, it must be specified for all backups in the backup set.

  If compression is used on the on the full backup, compression must be used on the incremental backups. Different compression levels are allowed for the backups in the backup set. For a backup, the default is compression level 1.

If you try to add an incremental backup to a backup set, the backup operation fails if the `gpbackup` options are not consistent.

For information about the `gpbackup` and `gprestore` utility options, see the *gpbackup* and *gprestore* reference documentation.

Example Using Incremental Backup Sets

Each backup has a timestamp taken when the backup is created. For example, if you create a backup on May 14, 2017, the backup file names contain `20170514`*`hhmmss`*. The *hhmmss* represents the time: hour, minute, and second.

This example assumes that you have created two full backups and incremental backups of the database *mytest*. To create the full backups, you used this command:

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data
```

You created incremental backups with this command:

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --
incremental
```

When you specify the `--backup-dir` option, the backups are created in the `/mybackup` directory on each Greenplum Database host.

In the example, the full backups have the timestamp keys `20170514054532` and `20171114064330`. The other backups are incremental backups. The example consists of two backup sets, the first with two

incremental backups, and second with one incremental backup. The backups are listed from earliest to most recent.

- 20170514054532 (full backup)
- 20170714095512
- 20170914081205
- 20171114064330 (full backup)
- 20180114051246

To create a new incremental backup based on the latest incremental backup, you must include the same `--backup-dir` option as the incremental backup as well as the options `--leaf-partition-data` and `--incremental`.

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --
incremental
```

You can specify the `--from-timestamp` option to create an incremental backup based on an existing incremental or full backup. Based on the example, this command adds a fourth incremental backup to the backup set that includes 20170914081205 as an incremental backup and uses 20170514054532 as the full backup.

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --
incremental --from-timestamp 20170914081205
```

This command creates an incremental backup set based on the full backup 20171114064330 and is separate from the backup set that includes the incremental backup 20180114051246.

```
gpbackup --dbname mytest --backup-dir /mybackup --leaf-partition-data --
incremental --from-timestamp 20171114064330
```

To restore a database with the incremental backup 20170914081205, you need the incremental backups 20120914081205 and 20170714095512, and the full backup 20170514054532. This would be the `gprestore` command.

```
gprestore --backup-dir /backupdir --timestamp 20170914081205
```

Creating an Incremental Backup with gpbackup

The `gpbackup` output displays the timestamp of the backup on which the incremental backup is based. In this example, the incremental backup is based on the backup with timestamp 20180802171642. The backup 20180802171642 can be an incremental or full backup.

```
$ gpbackup --dbname test --backup-dir /backups --leaf-partition-data --
incremental
20180803:15:40:51 gpbackup:gpadmin:mdw:002907-[INFO]:-Starting backup of
 database test
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Backup Timestamp =
 20180803154051
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Backup Database = test
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Gathering list of
 tables for backup
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Acquiring ACCESS SHARE
 locks on tables
Locks acquired:  5 / 5
 [================================================================] 100.00%
 0s
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Gathering additional
 table metadata
```

```
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Metadata will
 be written to /backups/gpseg-1/backups/20180803/20180803154051/
gpbackup_20180803154051_metadata.sql
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Writing global
 database metadata
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Global database
 metadata backup complete
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Writing pre-data
 metadata
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Pre-data metadata
 backup complete
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Writing post-data
 metadata
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Post-data metadata
 backup complete
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Basing incremental
 backup off of backup with timestamp = 20180802171642
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Writing data to file
Tables backed up:  4 / 4
 [==============================================================] 100.00% 0s
20180803:15:40:52 gpbackup:gpadmin:mdw:002907-[INFO]:-Data backup complete
20180803:15:40:53 gpbackup:gpadmin:mdw:002907-[INFO]:-Found neither /
usr/local/greenplum-db/./bin/gp_email_contacts.yaml nor /home/gpadmin/
gp_email_contacts.yaml
20180803:15:40:53 gpbackup:gpadmin:mdw:002907-[INFO]:-Email containing
 gpbackup report /backups/gpseg-1/backups/20180803/20180803154051/
gpbackup_20180803154051_report will not be sent
20180803:15:40:53 gpbackup:gpadmin:mdw:002907-[INFO]:-Backup completed
 successfully
```

Restoring from an Incremental Backup with gprestore

When restoring an from an incremental backup, you can specify the `--verbose` option to display the backups that are used in the restore operation on the command line. For example, the following `gprestore` command restores a backup using the timestamp `20180807092740`, an incremental backup. The output includes the backups that were used to restore the database data.

```
$ gprestore --create-db --timestamp 20180807162904 --verbose
...
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[INFO]:-Pre-data metadata
 restore complete
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Verifying backup
 file count
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Restoring data from
 backup with timestamp: 20180807162654
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for
 table public.tbl_ao from file (table 1 of 1)
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Checking whether
 segment agents had errors during restore
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Restoring data from
 backup with timestamp: 20180807162819
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for
 table public.test_ao from file (table 1 of 1)
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Checking whether
 segment agents had errors during restore
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Restoring data from
 backup with timestamp: 20180807162904
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for
 table public.homes2 from file (table 1 of 4)
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for
 table public.test2 from file (table 2 of 4)
```

```
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for
 table public.homes2a from file (table 3 of 4)
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Reading data for
 table public.test2a from file (table 4 of 4)
20180807:16:31:56 gprestore:gpadmin:mdw:008603-[DEBUG]:-Checking whether
 segment agents had errors during restore
20180807:16:31:57 gprestore:gpadmin:mdw:008603-[INFO]:-Data restore complete
20180807:16:31:57 gprestore:gpadmin:mdw:008603-[INFO]:-Restoring post-data
 metadata
20180807:16:31:57 gprestore:gpadmin:mdw:008603-[INFO]:-Post-data metadata
 restore complete
...
```

The output shows that the restore operation used three backups.

When restoring an from an incremental backup, `gprestore` also lists the backups that are used in the restore operation in the `gprestore` log file.

During the restore operation, `gprestore` displays an error if the full backup or other required incremental backup is not available.

Incremental Backup Notes

To create an incremental backup, or to restore data from an incremental backup set, you need the complete backup set. When you archive incremental backups, the complete backup set must be archived. You must archive all the files created on the master and all segments.

Each time `gpbackup` runs, the utility adds backup information to the history file `gpbackup_history.yaml` in the Greenplum Database master data directory. The file includes backup options and other backup information.

If you do not specify the `--from-timestamp` option when you create an incremental backup, `gpbackup` uses the most recent backup with a consistent set of options. The utility checks the backup history file to find the backup with a consistent set of options. If the utility cannot find a backup with a consistent set of options or the history file does not exist, `gpbackup` displays a message stating that a full backup must be created before an incremental can be created.

If you specify the `--from-timestamp` option when you create an incremental backup, `gpbackup` ensures that the options of the backup that is being created are consistent with the options of the specified backup.

The `gpbackup` option `--with-stats` is not required to be the same for all backups in the backup set. However, to perform a restore operation with the `gprestore` option `--with-stats` to restore statistics, the backup you specify must have must have used the `--with-stats` when creating the backup.

You can perform a restore operation from any backup in the backup set. However, changes captured in incremental backups later than the backup use to restore database data will not be restored.

When restoring from an incremental backup set, `gprestore` checks the backups and restores each append-optimized table from the most recent version of the append-optimized table in the backup set and restores the heap tables from the latest backup.

The incremental back up set, a full backup and associated incremental backups, must be on a single device. For example, the backups in a backup set must all be on a file system or must all be on a Data Domain system.

> **Warning:** Changes to the Greenplum Database segment configuration invalidate incremental backups. After you change the segment configuration (add or remove segment instances), you must create a full backup before you can create an incremental backup.

Using gpbackup and gprestore with BoostFS

You can use the Greenplum Database `gpbackup` and `gprestore` utilities with the Data Domain DD Boost File System Plug-In (BoostFS) to access a Data Domain system. BoostFS leverages DD Boost technology and helps reduce bandwidth usage, can improve backup-times, offers load-balancing and in-flight encryption, and supports the Data Domain multi-tenancy feature set.

You install the BoostFS plug-in on the Greenplum Database host systems to provide access to a Data Domain system as a standard file system mount point. With direct access to a BoostFS mount point, `gpbackup` and `gprestore` can leverage the storage and network efficiencies of the DD Boost protocol for backup and recovery.

For information about configuring BoostFS, you can download the *BoostFS for Linux Configuration Guide* from the Dell support site *https://www.dell.com/support* (requires login). After logging into the support site, you can find the guide by searching for "`BoostFS for Linux Configuration Guide`". You can limit your search results by choosing to list only `Manuals & Documentation` as resources.

To back up or restore with BoostFS, you include the option `--backup-dir` with the `gpbackup` or `gprestore` command to access the Data Domain system.

Installing BoostFS

Download the latest BoostFS RPM from the Dell support site *https://www.dell.com/support* (requires login).

After logging into the support site, you can find the RPM by searching for "`boostfs`". You can limit your search results by choosing to list only `Downloads & Drivers` as resources. To list the most recent RPM near the top of your search results, sort your results by descending date.

The RPM supports both RHEL and SuSE.

These steps install BoostFS and create a mounted directory that accesses a Data Domain system.

Perform the steps on all Greenplum Database hosts. The mounted directory you create must be the same on all hosts.

1. Copy the BoostFS RPM to the host and install the RPM.

   After installation, the DDBoostFS package files are located under `/opt/emc/boostfs`.
2. Set up the BoostFS lockbox with the storage unit with the `boostfs` utility. Enter the Data Domain user password at the prompts.

   ```
   /opt/emc/boostfs/bin/boostfs lockbox set -d <Data_Domain_IP> -s
    <Storage_Unit> -u <Data_Domain_User>
   ```

   The <*Storage_Unit*> is the Data Domain storage unit ID. The <*Data_Domain_User*> is a Data Domain user with access to the storage unit.
3. Create the directory in the location you want to mount BoostFS.

   ```
   mkdir <path_to_mount_directory>
   ```
4. Mount the Data Domain storage unit with the `boostfs` utility. Use the `mount` option `-allow-others=true` to allow other users to write to the BoostFS mounted file system.

   ```
   /opt/emc/boostfs/bin/boostfs mount <path_to_mount_directory> -d
    $<Data_Domain_IP> -s <Storage_Unit> -o allow-others=true
   ```
5. Confirm that the mount was successful by running this command.

   ```
   mountpoint <mounted_directory>
   ```

The command lists the directory as a mount point.

```
<mounted_directory> is a mountpoint
```

You can now run `gpbackup` and `gprestore` with the `--backup-dir` option to back up a database to *<mounted_directory>* on the Data Domain system and restore data from the Data Domain system.

Backing Up and Restoring with BoostFS

These are required `gpbackup` options when backing up data to a Data Domain system with BoostFS.

- `--backup-dir` - Specify the mounted Data Domain storage unit.
- `--no-compression` - Disable compression. Data compression interferes with DD Boost data de-duplication.
- `--single-data-file` - Create a single data file on each segment host. A single data file avoids a BoostFS stream limitation.

When you use `gprestore` to restore a backup from a Data Domain system with BoostFS, you must specify the mounted Data Domain storage unit with the option `--backup-dir`.

When you use the `gpbackup` option `--single-data-file`, you cannot specify the `--jobs` option to perform a parallel restore operation with `gprestore`.

This example `gpbackup` command backs up the database `test`. The example assumes that the directory `/boostfs-test` is the mounted Data Domain storage unit.

```
$ gpbackup --dbname test --backup-dir /boostfs-test/ --single-data-file --
no-compression
```

These commands drop the database `test` and restore the database from the backup.

```
$ dropdb test
$ gprestore --backup-dir /boostfs-test/ --timestamp 20171103153156 --create-
db
```

The value `20171103153156` is the timestamp of the `gpbackup` backup set to restore. For information about how `gpbackup` uses timesamps when creating backups, see *Parallel Backup with gpbackup and gprestore*. For information about the `-timestamp` option, see *gprestore*.

Using gpbackup Storage Plugins

You can configure the Greenplum Database *`gpbackup`* and *`gprestore`* utilities to use a storage plugin to process backup files during a backup or restore operation. For example, during a backup operation, the plugin sends the backup files to a remote location. During a restore operation, the plugin retrieves the files from the remote location.

You can also develop a custom storage plugin with the Greenplum Database Backup/Restore Storage Plugin API (Beta). See *Backup/Restore Storage Plugin API (Beta)*.

> **Note:** Only the Backup/Restore Storage Plugin API is a Beta feature. The storage plugins are supported features.

Using the S3 Storage Plugin with gpbackup and gprestore

The S3 storage plugin application lets you use an Amazon Simple Storage Service (Amazon S3) location to store and retrieve backups when you run *`gpbackup`* and *`gprestore`*. Amazon S3 provides secure, durable, highly-scalable object storage.

The S3 storage plugin can also connect to an Amazon S3 compatible service such as *Dell EMC Elastic Cloud Storage* (ECS) and *Minio*.

To use the S3 storage plugin application, you specify the location of the plugin and the S3 login and backup location in a configuration file. When you run `gpbackup` or `gprestore`, you specify the configuration file with the option `--plugin-config`. For information about the configuration file, see *S3 Storage Plugin Configuration File Format*.

If you perform a backup operation with the `gpbackup` option `--plugin-config`, you must also specify the `--plugin-config` option when you restore the backup with `gprestore`.

S3 Storage Plugin Configuration File Format

The configuration file specifies the absolute path to the Greenplum Database S3 storage plugin executable, connection credentials, and S3 location.

The S3 storage plugin configuration file uses the *YAML 1.1* document format and implements its own schema for specifying the location of the Greenplum Database S3 storage plugin, connection credentials, and S3 location and login information.

The configuration file must be a valid YAML document. The `gpbackup` and `gprestore` utilities process the control file document in order and use indentation (spaces) to determine the document hierarchy and the relationships of the sections to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.

This is the structure of a S3 storage plugin configuration file.

```
executablepath: <absolute-path-to-gpbackup_s3_plugin>
options:
  region: <aws-region>
  endpoint: <S3-endpoint>
  aws_access_key_id: <aws-user-id>
  aws_secret_access_key: <aws-user-id-key>
  bucket: <s3-bucket>
  folder: <s3-location>
  encryption: [on|off]
```

`executablepath`

Required. Absolute path to the plugin executable. For example, the Pivotal Greenplum Database installation location is `$GPHOME/bin/gpbackup_s3_plugin`. The plugin must be in the same location on every Greenplum Database host.

`options`

Required. Begins the S3 storage plugin options section.

`region`

Required for AWS S3. If connecting to an S3 compatible service, this option is not required.

`endpoint`

Required for an S3 compatible service. Specify this option to connect to an S3 compatible service such as ECS. The plugin connects to the specified S3 endpoint (hostname or IP address) to access the S3 compatible data store.

If this option is specified, the plugin ignores the `region` option and does not use AWS to resolve the endpoint. When this option is not specified, the plugin uses the `region` to determine AWS S3 endpoint.

`aws_access_key_id`

Optional. The S3 ID to access the S3 bucket location that stores backup files.

If this parameter is not specified, S3 authentication information from the session environment is used. See *Notes*.

`aws_secret_access_key`

Required only if you specify `aws_access_key_id`. The S3 passcode for the S3 ID to access the S3 bucket location.

`bucket`

Required. The name of the S3 bucket in the AWS region or S3 compatible data store. The bucket must exist.

`folder`

Required. The S3 location for backups. During a backup operation, the plugin creates the S3 location if it does not exist in the S3 bucket.

`encryption`

Optional. Enable or disable use of Secure Sockets Layer (SSL) when connecting to an S3 location. Default value is `on`, use connections that are secured with SSL. Set this option to `off` to connect to an S3 compatible service that is not configured to use SSL.

Any value other than `off` is treated as `on`.

Example

This is an example S3 storage plugin configuration file that is used in the next `gpbackup` example command. The name of the file is `s3-test-config.yaml`.

```
executablepath: $GPHOME/bin/gpbackup_s3_plugin
options:
  region: us-west-2
  aws_access_key_id: test-s3-user
  aws_secret_access_key: asdf1234asdf
  bucket: gpdb-backup
  folder: test/backup3
```

This `gpbackup` example backs up the database demo using the S3 storage plugin. The absolute path to the S3 storage plugin configuration file is `/home/gpadmin/s3-test`.

```
gpbackup --dbname demo --plugin-config /home/gpadmin/s3-test-config.yaml
```

The S3 storage plugin writes the backup files to this S3 location in the AWS region us-west-2.

```
gpdb-backup/test/backup3/backups/YYYYMMDD/YYYYMMDDHHMMSS/
```

Notes

The S3 storage plugin application must be in the same location on every Greenplum Database host. The configuration file is required only on the master host.

When you perform a backup with the S3 storage plugin, the plugin stores the backup files in this location in the S3 bucket.

```
<folder>/backups/<datestamp>/<timestamp>
```

Where *folder* is the location you specified in the S3 configuration file, and *datestamp* and *timestamp* are the backup date and time stamps.

Using Amazon S3 to back up and restore data requires an Amazon AWS account with access to the Amazon S3 bucket. These are the Amazon S3 bucket permissions required for backing up and restoring data.

- `Upload/Delete` for the S3 user ID that uploads the files
- `Open/Download` and `View` for the S3 user ID that accesses the files

If `aws_access_key_id` and `aws_secret_access_key` are not specified in the configuration file, the S3 plugin uses S3 authentication information from the system environment of the session running the backup operation. The S3 plugin searches for the information in these sources, using the first available source.

1. The environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.
2. The authentication information set with the AWS CLI command `aws configure`.
3. The credentials of the Amazon EC2 IAM role if the backup is run from an EC2 instance.

For information about Amazon S3, see *Amazon S3*.

- For information about Amazon S3 regions and endpoints, see *http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region*.
- For information about S3 buckets and folders, see the Amazon S3 documentation *https://aws.amazon.com/documentation/s3/*.

Using the DD Boost Storage Plugin with gpbackup, gprestore, and gpbackup_manager

> **Note:** The DD Boost storage plugin is available in the commercial release of Pivotal Greenplum Backup and Restore.

Dell EMC Data Domain Boost (DD Boost) is Dell EMC software that can be used with the *`gpbackup`* and *`gprestore`* utilities to perform faster backups to the Dell EMC Data Domain storage appliance. You can also replicate a backup on a separate, remote Data Domain system for disaster recovery with gpbackup or *gpbackup_manager*. For information about replication, see *Replicating Backups*.

To use the DD Boost storage plugin application, you first create a configuration file to specify the location of the plugin, the DD Boost login, and the backup location. When you run `gpbackup` or `gprestore`, you specify the configuration file with the option `--plugin-config`. For information about the configuration file, see *DD Boost Storage Plugin Configuration File Format*.

If you perform a backup operation with the `gpbackup` option `--plugin-config`, you must also specify the `--plugin-config` option when you restore the backup with `gprestore`.

DD Boost Storage Plugin Configuration File Format

The configuration file specifies the absolute path to the Greenplum Database DD Boost storage plugin executable, DD Boost connection credentials, and Data Domain location. The configuration file is required only on the master host. The DD Boost storage plugin application must be in the same location on every Greenplum Database host.

The DD Boost storage plugin configuration file uses the *YAML 1.1* document format and implements its own schema for specifying the DD Boost information.

The configuration file must be a valid YAML document. The `gpbackup` and `gprestore` utilities process the configuration file document in order and use indentation (spaces) to determine the document hierarchy and the relationships of the sections to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.

This is the structure of a DD Boost storage plugin configuration file.

```
executablepath: <absolute-path-to-gpbackup_ddboost_plugin>
options:
  hostname: "<data-domain-host>"
  username: "<ddboost-ID>"
  password-encryption: "on" | "off"
  password: "<ddboost-pwd>"
  storage_unit: "<data-domain-id>"
  directory: "<data-domain-dir>"
  replication: "on" | "off"
  replication-streams: integer
  remote_hostname: "<remote-dd-host>"
  remote_username: "<remote-ddboost-ID>"
```

```
  remote_password_encryption "on" | "off"
  remote_password: "<remote-dd-pwd>"
  remote_storage_unit: "<remote-dd-ID>"
  remote_directory: "<remote-dd-dir>"
```

executablepath

Required. Absolute path to the plugin executable. For example, the Pivotal Greenplum Database installation location is `$GPHOME/bin/gpbackup_ddboost_plugin`. The plugin must be in the same location on every Greenplum Database host.

options

Required. Begins the DD Boost storage plugin options section.

hostname

Required. The IP address or hostname of the host. There is a 30-character limit.

username

Required. The Data Domain Boost user name. There is a 30-character limit.

password_encryption

Optional. Specifies whether the `password` option value is encrypted. Default value is `off`. Use the *gpbackup_manager* `encrypt-password` command to encrypt the plain-text password for the DD Boost user. If the `replication` option is `on`, `gpbackup_manager` also encrypts the remote Data Domain user's password. Copy the encrypted password(s) from the `gpbackup_manager` output to the `password` options in the configuration file.

password

Required. The passcode for the DD Boost user to access the Data Domain storage unit. If the `password_encryption` option is `on`, this is an encrypted password.

storage-unit

Required. A valid storage unit name for the Data Domain system that is used for backup and restore operations.

directory

Required. The location for the backup files, configuration files, and global objects on the Data Domain system. The location on the system is */<data-domain-dir>* in the storage unit of the system.

During a backup operation, the plugin creates the directory location if it does not exist in the storage unit and stores the backup in this directory */<data-domain-dir>/YYYYMMDD/YYYYMMDDHHMMSS/*.

replication

Optional. Enables or disables backup replication with DD Boost managed file replication when `gpbackup` performs a backup operation. Value is either `on` or `off`. Default value is `off`, backup replication is disabled. When the value is `on`, the DD Boost plugin replicates the backup on the Data Domain system that you specify with the `remote_*` options.

The `replication` option and `remote_*` options are ignored when performing a restore operation with `gprestore`. The `remote_*` options are ignored if `replication` is `off`.

This option is ignored when you perform replication with the `gpbackup_manager replicate-backup` command. For information about replication,see *Replicating Backups*.

replication-streams

Optional. Used with the `gpbackup_manager replicate-backup` command, ignored otherwise. Specifies the maximum number of Data Domain I/O streams that can be used when replicating a backup set on a remote Data Domain server from the Data Domain server that contains the backup. Default value is 1.

This option is ignored when you perform replication with `gpbackup`. The default value is used.

`remote_hostname`

Required when performing replication. The IP address or hostname of the Data Domain system that is used for remote backup storage. There is a 30-character limit.

`remote_username`

Required when performing replication. The Data Domain Boost user name that accesses the remote Data Domain system. There is a 30-character limit.

`remote_password_encryption`

Optional when performing replication. Specifies whether the `remote_password` option value is encrypted. The default value is `off`. To set up password encryption use the *`gpbackup_manager`* `encrypt-password` command to encrypt the plain-text passwords for the DD Boost user. If the `replication` parameter is `on`, `gpbackup_manager` also encrypts the remote Data Domain user's password. Copy the encrypted passwords from the `gpbackup_manager` output to the password options in the configuration file.

`remote_password`

Required when performing replication. The passcode for the DD Boost user to access the Data Domain storage unit on the remote system. If the `remote_password_encryption` option is `on`, this is an encrypted password.

`remote_storage_unit`

Required when performing replication. A valid storage unit name for the remote Data Domain system that is used for backup replication.

`remote_directory`

Required when performing replication. The location for the replicated backup files, configuration files, and global objects on the remote Data Domain system. The location on the system is */<remote-dd-dir>* in the storage unit of the remote system.

During a backup operation, the plugin creates the directory location if it does not exist in the storage unit of the remote Data Domain system and stores the replicated backup in this directory */<remote-dd-dir>/YYYYMMDD/YYYYMMDDHHMMSS/*.

Examples

This is an example DD Boost storage plugin configuration file that is used in the next `gpbackup` example command. The name of the file is `ddboost-test-config.yaml`.

```
executablepath: $GPHOME/bin/gpbackup_ddboost_plugin
options:
  hostname: "192.0.2.230"
  username: "test-ddb-user"
  password: "asdf1234asdf"
  storage_unit: "gpdb-backup"
  directory: "test/backup"
```

This `gpbackup` example backs up the database demo using the DD Boost storage plugin. The absolute path to the DD Boost storage plugin configuration file is `/home/gpadmin/ddboost-test-config.yml`.

```
gpbackup --dbname demo --single-data-file --plugin-config /home/gpadmin/
ddboost-test-config.yaml
```

The DD Boost storage plugin writes the backup files to this directory of the Data Domain storage unit `gpdb-backup`.

```
/test/backup/YYYYMMDD/YYYYMMDDHHMMSS/
```

This is an example DD Boost storage plugin configuration file that enables replication.

```
executablepath: $GPHOME/bin/gpbackup_ddboost_plugin
options:
  hostname: "192.0.2.230"
  username: "test-ddb-user"
  password: "asdf1234asdf"
  storage_unit: "gpdb-backup"
  directory: "test/backup"
  replication: "on"
  remote_hostname: "192.0.3.20"
  remote_username: "test-dd-remote"
  remote_password: "qwer2345erty"
  remote_storage_unit: "gpdb-remote"
  remote_directory: "test/replication"
```

To restore from the replicated backup in the previous example, you can run `gprestore` with the DD Boost storage plugin and specify a configuration file with this information.

```
executablepath: $GPHOME/bin/gpbackup_ddboost_plugin
options:
  hostname: "192.0.3.20"
  remote_username: "test-dd-remote"
  remote_password: "qwer2345erty"
  storage_unit: "gpdb-remote"
  directory: "test/replication"
```

Notes

Dell EMC DD Boost is integrated with Pivotal Greenplum Database and requires a DD Boost license. Open source Greenplum Database cannot use the DD Boost software, but can back up to a Dell EMC Data Domain system mounted as an NFS share on the Greenplum master and segment hosts.

When you perform a backup with the DD Boost storage plugin, the plugin stores the backup files in this location in the Data Domain storage unit.

```
<directory>/backups/<datestamp>/<timestamp>
```

Where <*directory*> is the location you specified in the DD Boost configuration file, and <*datestamp*> and <*timestamp*> are the backup date and time stamps.

Replicating Backups

You can use *gpbackup* or *gpbackup_manager* with the DD Boost storage plugin to replicate a backup from one Data Domain system to a second, remote, Data Domain system for disaster recovery. You can replicate a backup as part of the backup process, or replicate an existing backup set as a separate operation. Both methods require a *DD Boost configuration file* that includes options that specify Data Domain system locations and DD Boost configuration. The DD Boost storage plugin replicates the backup set on the remote Data Domain system with DD Boost managed file replication.

When replicating a backup, the Data Domain system where the backup is stored must have access to the remote Data Domain system where the replicated backup is stored.

To restore data from a replicated backup, use *gprestore* with the DD Boost storage plugin. In the configuration file, specify the location of the backup in the DD Boost configuration file.

For example configuration files, see *Examples* in *Using the DD Boost Storage Plugin with gpbackup, gprestore, and gpbackup_manager*.

Replicate a Backup as Part of the Backup Process

Use the `gpbackup` utility to replicate a backup set as part of the backup process.

To enable replication during a back up, add the backup replication options to the configuration file. Set the configuration file `replication` option to `on` and add the options that the plugin uses to access the remote Data Domain system that stores the replicated backup.

When using `gpbackup`, the `replication` option must be set to `on`.

The configuration file `replication-streams` option is ignored, the default value is used.

Performing a backup operation with replication increases the time required to perform a backup. The backup set is copied to the local Data Domain system, and then replicated on the remote Data Domain system using DD Boost managed file replication. The backup operation completes after the backup set is replicated on the remote system.

Replicate an Existing Backup

Use the `gpbackup_manager replicate-backup` command to replicate an existing backup set that is on a Data Domain system and was created by `gpbackup`.

When you run `backup_manager replicate-backup`, specify a DD Boost configuration file that contains the same type of information that is in the configuration file used to replicate a backup set with `gpbackup`.

When using the `gpbackup_manager replicate-backup` command, the configuration file `replication` option is ignored. The command always attempts to replicate a back up.

Backup/Restore Storage Plugin API (Beta)

This topic describes how to develop a custom storage plugin with the Greenplum Database Backup/Restore Storage Plugin API.

> **Note:** Only the Backup/Restore Storage Plugin API is a Beta feature. The storage plugins are supported features.

The Backup/Restore Storage Plugin API provides a framework that you can use to develop and integrate a custom backup storage system with the Greenplum Database *`gpbackup`*, *`gpbackup_manager`*, and *`gprestore`* utilities.

The Backup/Restore Storage Plugin API defines a set of interfaces that a plugin must support. The API also specifies the format and content of a configuration file for a plugin.

When you use the Backup/Restore Storage Plugin API, you create a plugin that the Greenplum Database administrator deploys to the Greenplum Database cluster. Once deployed, the plugin is available for use in certain backup and restore operations.

This topic includes the following subtopics:

- *Plugin Configuration File*
- *Plugin API*
- *Plugin Commands*
- *Implementing a Backup/Restore Storage Plugin*
- *Verifying a Backup/Restore Storage Plugin*
- *Packaging and Deploying a Backup/Restore Storage Plugin*

Plugin Configuration File

Specifying the `--plugin-config` option to the `gpbackup` and `gprestore` commands instructs the utilities to use the plugin specified in the configuration file for the operation.

The plugin configuration file provides information for both Greenplum Database and the plugin. The Backup/Restore Storage Plugin API defines the format of, and certain keywords used in, the plugin configuration file.

A plugin configuration file is a YAML file in the following format:

```
executablepath: path_to_plugin_executable
options:
  keyword1: value1
  keyword2: value2
  ...
  keywordN: valueN
```

`gpbackup` and `gprestore` use the **`executablepath`** value to determine the file system location of the plugin executable program.

The plugin configuration file may also include keywords and values specific to a plugin instance. A backup/restore storage plugin can use the **`options`** block specified in the file to obtain information from the user that may be required to perform its tasks. This information may include location, connection, or authentication information, for example. The plugin should both specify and consume the content of this information in *`keyword`*`:`*`value`* syntax.

A sample plugin configuration file for the Greenplum Database S3 backup/restore storage plugin follows:

```
executablepath: $GPHOME/bin/gpbackup_s3_plugin
options:
  region: us-west-2
  aws_access_key_id: notarealID
  aws_secret_access_key: notarealkey
  bucket: gp_backup_bucket
  folder: greenplum_backups
```

Plugin API

The plugin that you implement when you use the Backup/Restore Storage Plugin API is an executable program that supports specific *command*s invoked by `gpbackup` and `gprestore` at defined points in their respective life cycle operations:

- The Greenplum Database Backup/Restore Storage Plugin API provides hooks into the `gpbackup` lifecycle at initialization, during backup, and at cleanup/exit time.
- The API provides hooks into the `gprestore` lifecycle at initialization, during restore, and at cleanup/exit time.
- The API provides arguments that specify the execution scope (master host, segment host, or segment instance) for a plugin setup or cleanup command. The scope can be one of these values.
  - `master` - Execute the plugin once on the master host.
  - `segment_host` - Execute the plugin once on each of the segment hosts.
  - `segment` - Execute the plugin once for each active segment instance on the host running the segment instance.

  The Greenplum Database hosts and segment instances are based on the Greenplum Database configuration when the back up started. The values `segment_host` and `segment` are provided as a segment host can host multiple segment instances. There might be some setup or cleanup required at the segment host level as compared to each segment instance.

The Plugin API also defines the `delete_backup` command, which is called by the `gpbackup_manager` utility. (The `gpbackup_manager` source code is proprietary and the utility is available only in the Pivotal Greenplum Backup and Restore download from *Pivotal Network*.)

The Backup/Restore Storage Plugin API defines the following call syntax for a backup/restore storage plugin executable program:

```
plugin_executable command config_file args
```

where:

- *plugin_executable* - The absolute path of the backup/restore storage plugin executable program. This path is determined by the `executablepath` property value configured in the plugin's configuration YAML file.
- *command* - The name of a Backup/Restore Storage Plugin API command that identifies a specific entry point to a `gpbackup` or `gprestore` lifecycle operation.
- *config_file* - The absolute path of the plugin's configuration YAML file.
- *args* - The command arguments; the actual arguments differ depending upon the *command* specified.

Plugin Commands

The Greenplum Database Backup/Restore Storage Plugin API defines the following commands:

Table 23: Backup/Restore Storage Plugin API Commands

Command Name	Description
plugin_api_version	Return the version of the Backup/Restore Storage Plugin API supported by the plugin. The currently supported version is 0.4.0.
setup_plugin_for_backup	Initialize the plugin for a backup operation.
backup_file	Move a backup file to the remote storage system.
backup_data	Move streaming data from `stdin` to a file on the remote storage system.
delete_backup	Delete the directory specified by the given backup timestamp on the remote system.
cleanup_plugin_for_backup	Clean up after a backup operation.
setup_plugin_for_restore	Initialize the plugin for a restore operation.
restore_file	Move a backup file from the remote storage system to a designated location on the local host.
restore_data	Move a backup file from the remote storage system, streaming the data to `stdout`.
cleanup_plugin_for_restore	Clean up after a restore operation.

A backup/restore storage plugin must support every command identified above, even if it is a no-op.

Implementing a Backup/Restore Storage Plugin

You can implement a backup/restore storage plugin executable in any programming or scripting language.

The tasks performed by a backup/restore storage plugin will be very specific to the remote storage system. As you design the plugin implementation, you will want to:

- Examine the connection and data transfer interface to the remote storage system.
- Identify the storage path specifics of the remote system.
- Identify configuration information required from the user.
- Define the keywords and value syntax for information required in the plugin configuration file.

- Determine if, and how, the plugin will modify (compress, etc.) the data en route to/from the remote storage system.
- Define a mapping between a `gpbackup` file path and the remote storage system.
- Identify how `gpbackup` options affect the plugin, as well as which are required and/or not applicable. For example, if the plugin performs its own compression, `gpbackup` must be invoked with the `--no-compression` option to prevent the utility from compressing the data.

A backup/restore storage plugin that you implement must:

- Support all plugin commands identified in *Plugin Commands*. Each command must exit with the values identified on the command reference page.

Refer to the *gpbackup-s3-plugin* github repository for an example plugin implementation.

Verifying a Backup/Restore Storage Plugin

The Backup/Restore Storage Plugin API includes a test bench that you can run to ensure that a plugin is well integrated with `gpbackup` and `gprestore`.

The test bench is a `bash` script that you run in a Greenplum Database installation. The script generates a small (<1MB) data set in a Greenplum Database table, explicitly tests each command, and runs a backup and restore of the data (file and streaming). The test bench invokes `gpbackup` and `gprestore`, which in turn individually call/test each Backup/Restore Storage Plugin API command implemented in the plugin.

The test bench program calling syntax is:

```
plugin_test_bench.sh plugin_executable plugin_config
```

Procedure

To run the Backup/Restore Storage Plugin API test bench against a plugin:

1. Log in to the Greenplum Database master host and set up your environment. For example:

   ```
   $ ssh gpadmin@<gpmaster>
   gpadmin@gpmaster$ . /usr/local/greenplum-db/greenplum_path.sh
   ```

2. Obtain a copy of the test bench from the `gpbackup` github repository. For example:

   ```
   $ git clone git@github.com:greenplum-db/gpbackup.git
   ```

   The clone operation creates a directory named `gpbackup/` in the current working directory.
3. Locate the test bench program in the `gpbackup/master/plugins` directory. For example:

   ```
   $ ls gpbackup/master/plugins/plugin_test_bench.sh
   ```

4. Copy the plugin executable program and the plugin configuration YAML file from your development system to the Greenplum Database master host. Note the file system location to which you copied the files.
5. Copy the plugin executable program from the Greenplum Database master host to the same file system location on each segment host.
6. If required, edit the plugin configuration YAML file to specify the absolute path of the plugin executable program that you just copied to the Greenplum segments.
7. Run the test bench program against the plugin. For example:

   ```
   $ gpbackup/master/plugins/plugin_test_bench.sh /path/to/pluginexec /path/
   to/plugincfg.yaml
   ```

8. Examine the test bench output. Your plugin passed the test bench if all output messages specify `RUNNING` and `PASSED`. For example:

```
# ----------------------------------------------
# Starting gpbackup plugin tests
# ----------------------------------------------
[RUNNING] plugin_api_version
[PASSED] plugin_api_version
[RUNNING] setup_plugin_for_backup
[RUNNING] backup_file
[RUNNING] setup_plugin_for_restore
[RUNNING] restore_file
[PASSED] setup_plugin_for_backup
[PASSED] backup_file
[PASSED] setup_plugin_for_restore
[PASSED] restore_file
[RUNNING] backup_data
[RUNNING] restore_data
[PASSED] backup_data
[PASSED] restore_data
[RUNNING] cleanup_plugin_for_backup
[PASSED] cleanup_plugin_for_backup
[RUNNING] cleanup_plugin_for_restore
[PASSED] cleanup_plugin_for_restore
[RUNNING] gpbackup with test database
[RUNNING] gprestore with test database
[PASSED] gpbackup and gprestore
# ----------------------------------------------
# Finished gpbackup plugin tests
# ----------------------------------------------
```

Packaging and Deploying a Backup/Restore Storage Plugin

Your backup/restore storage plugin is ready to be deployed to a Greenplum Database installation after the plugin passes your testing and the test bench verification. When you package the backup/restore storage plugin, consider the following:

- The backup/restore storage plugin must be installed in the same file system location on every host in the Greenplum Database cluster. Provide installation instructions for the plugin identifying the same.
- The `gpadmin` user must have permission to traverse the file system path to the backup/restore plugin executable program.
- Include a template configuration file with the plugin.
- Document the valid plugin configuration keywords, making sure to include the syntax of expected values.
- Document required `gpbackup` options and how they affect plugin processing.

backup_data

Plugin command to move streaming data from `stdin` to the remote storage system.

Synopsis

```
plugin_executable backup_data plugin_config_file data_filenamekey
```

Description

`gpbackup` invokes the `backup_data` plugin command on each segment host during a streaming backup.

The `backup_data` implementation should read a potentially large stream of data from `stdin` and write the data to a single file on the remote storage system. The data is sent to the command as a single

continuous stream per Greenplum Database segment. If `backup_data` modifies the data in any manner (i.e. compresses), `restore_data` must perform the reverse operation.

Name or maintain a mapping from the destination file to `data_filenamekey`. This will be the file key used for the restore operation.

Arguments

plugin_config_file

The absolute path to the plugin configuration YAML file.

data_filenamekey

The mapping key for a specially-named backup file for streamed data.

Exit Code

The `backup_data` command must exit with a value of 0 on success, non-zero if an error occurs. In the case of a non-zero exit code, `gpbackup` displays the contents of `stderr` to the user.

backup_file

Plugin command to move a backup file to the remote storage system.

Synopsis

```
plugin_executable backup_file plugin_config_file file_to_backup
```

Description

`gpbackup` invokes the `backup_file` plugin command on the master and each segment host for the file that `gpbackup` writes to a backup directory on local disk.

The `backup_file` implementation should process and copy the file to the remote storage system. Do not remove the local copy of the file that you specify with `file_to_backup`.

Arguments

plugin_config_file

The absolute path to the plugin configuration YAML file.

file_to_backup

The absolute path to a local backup file generated by `gpbackup`. Do not remove the local copy of the file that you specify with `file_to_backup`.

Exit Code

The `backup_file` command must exit with a value of 0 on success, non-zero if an error occurs. In the case of a non-zero exit code, `gpbackup` displays the contents of `stderr` to the user.

cleanup_plugin_for_backup

Plugin command to clean up a storage plugin after backup.

Synopsis

```
plugin_executable cleanup_plugin_for_backup plugin_config_file local_backup_dir scope
```

```
plugin_executable cleanup_plugin_for_backup plugin_config_file local_backup_dir scope co
```

Description

`gpbackup` invokes the `cleanup_plugin_for_backup` plugin command when a `gpbackup` operation completes, both in success and failure cases. The *scope* argument specifies the execution scope. `gpbackup` will invoke the command with each of the *scope* values.

The `cleanup_plugin_for_backup` command should perform the actions necessary to clean up the remote storage system after a backup. Clean up activities may include removing remote directories or temporary files created during the backup, disconnecting from the backup service, etc.

Arguments

`plugin_config_file`

The absolute path to the plugin configuration YAML file.

`local_backup_dir`

The local directory on the Greenplum Database host (master and segments) to which `gpbackup` wrote backup files.

- When *scope* is `master`, the *local_backup_dir* is the backup directory of the Greenplum Database master.
- When *scope* is `segment`, the *local_backup_dir* is the backup directory of a segment instance. The *contentID* identifies the segment instance.
- When the scope is `segment_host`, the *local_backup_dir* is an arbitrary backup directory on the host.

`scope`

The execution scope value indicates the host and number of times the plugin command is executed. *scope* can be one of these values:

- `master` - Execute the plugin command once on the master host.
- `segment_host` - Execute the plugin command once on each of the segment hosts.
- `segment` - Execute the plugin command once for each active segment instance on the host running the segment instance. The *contentID* identifies the segment instance.

The Greenplum Database hosts and segment instances are based on the Greenplum Database configuration when the back up was first initiated.

`contentID`

The contentID of the Greenplum Database master or segment instance corresponding to the scope. *contentID* is passed only when the *scope* is `master` or `segment`.

- When *scope* is `master`, the *contentID* is `-1`.
- When *scope* is `segment`, the *contentID* is the content identifier of an active segment instance.

Exit Code

The `cleanup_plugin_for_backup` command must exit with a value of 0 on success, non-zero if an error occurs. In the case of a non-zero exit code, `gpbackup` displays the contents of `stderr` to the user.

cleanup_plugin_for_restore

Plugin command to clean up a storage plugin after restore.

Synopsis

```
plugin_executable cleanup_plugin_for_restore plugin_config_file local_backup_dir scope
```

```
plugin_executable cleanup_plugin_for_restore plugin_config_file local_backup_dir scope c
```

Description

`gprestore` invokes the `cleanup_plugin_for_restore` plugin command when a `gprestore` operation completes, both in success and failure cases. The *scope* argument specifies the execution scope. `gprestore` will invoke the command with each of the *scope* values.

The `cleanup_plugin_for_restore` implementation should perform the actions necessary to clean up the remote storage system after a restore. Clean up activities may include removing remote directories or temporary files created during the restore, disconnecting from the backup service, etc.

Arguments

plugin_config_file

The absolute path to the plugin configuration YAML file.

local_backup_dir

The local directory on the Greenplum Database host (master and segments) from which `gprestore` reads backup files.

- When *scope* is `master`, the *local_backup_dir* is the backup directory of the Greenplum Database master.
- When *scope* is `segment`, the *local_backup_dir* is the backup directory of a segment instance. The *contentID* identifies the segment instance.
- When the scope is `segment_host`, the *local_backup_dir* is an arbitrary backup directory on the host.

scope

The execution scope value indicates the host and number of times the plugin command is executed. *scope* can be one of these values:

- `master` - Execute the plugin command once on the master host.
- `segment_host` - Execute the plugin command once on each of the segment hosts.
- `segment` - Execute the plugin command once for each active segment instance on the host running the segment instance. The *contentID* identifies the segment instance.

The Greenplum Database hosts and segment instances are based on the Greenplum Database configuration when the back up was first initiated.

contentID

The contentID of the Greenplum Database master or segment instance corresponding to the scope. *contentID* is passed only when the *scope* is `master` or `segment`.

- When *scope* is `master`, the *contentID* is `-1`.
- When *scope* is `segment`, the *contentID* is the content identifier of an active segment instance.

Exit Code

The `cleanup_plugin_for_restore` command must exit with a value of 0 on success, non-zero if an error occurs. In the case of a non-zero exit code, `gprestore` displays the contents of `stderr` to the user.

delete_backup

Plugin command to delete the directory for a given backup timestamp from a remote system.

Synopsis

```
delete_backup plugin_config_file timestamp
```

Description

`gpbackup_manager` invokes the `delete_backup` plugin command to delete the directory specified by the backup timestamp on the remote system.

Arguments

plugin_config_file

The absolute path to the plugin configuration YAML file.

timestamp

The timestamp for the backup to delete.

Exit Code

The `delete_backup` command must exit with a value of 0 on success, or a non-zero value if an error occurs. In the case of a non-zero exit code, `gpbackup_manager` displays the contents of `stderr` to the user.

Example

```
my_plugin delete_backup /home/my-plugin_config.yaml 20191208130802
```

plugin_api_version

Plugin command to display the supported Backup Storage Plugin API version.

Synopsis

```
plugin_executable plugin_api_version
```

Description

`gpbackup` and `gprestore` invoke the `plugin_api_version` plugin command before a backup or restore operation to determine Backup Storage Plugin API version compatibility.

Return Value

The `plugin_api_version` command must return the Backup Storage Plugin API version number supported by the storage plugin, "0.4.0".

restore_data

Plugin command to stream data from the remote storage system to `stdout`.

Synopsis

```
plugin_executable restore_data plugin_config_file data_filenamekey
```

Description

`gprestore` invokes the `restore_data` plugin command on each segment host when a restoring a streaming backup.

The `restore_data` implementation should read a potentially large data file named or mapped to `data_filenamekey` from the remote storage system and write the contents to `stdout`. If the `backup_data` command modified the data in any way (i.e. compressed), `restore_data` should perform the reverse operation.

Arguments

`plugin_config_file`

The absolute path to the plugin configuration YAML file.

`data_filenamekey`

The mapping key to a backup file on the remote storage system. *data_filenamekey* is the same key provided to the `backup_data` command.

Exit Code

The `restore_data` command must exit with a value of 0 on success, non-zero if an error occurs. In the case of a non-zero exit code, `gprestore` displays the contents of `stderr` to the user.

restore_file

Plugin command to move a backup file from the remote storage system.

Synopsis

```
plugin_executable restore_file plugin_config_file file_to_restore
```

Description

`gprestore` invokes the `restore_file` plugin command on the master and each segment host for the file that `gprestore` will read from a backup directory on local disk.

The `restore_file` command should process and move the file from the remote storage system to *`file_to_restore`* on the local host.

Arguments

`plugin_config_file`

The absolute path to the plugin configuration YAML file.

`file_to_restore`

The absolute path to which to move a backup file from the remote storage system.

Exit Code

The `restore_file` command must exit with a value of 0 on success, non-zero if an error occurs. In the case of a non-zero exit code, `gprestore` displays the contents of `stderr` to the user.

setup_plugin_for_backup

Plugin command to initialize a storage plugin for the backup operation.

Synopsis

```
plugin_executable setup_plugin_for_backup plugin_config_file local_backup_dir scope
```

```
plugin_executable setup_plugin_for_backup plugin_config_file local_backup_dir scope cont
```

Description

`gpbackup` invokes the `setup_plugin_for_backup` plugin command during `gpbackup` initialization phase. The *scope* argument specifies the execution scope. `gpbackup` will invoke the command with each of the *scope* values.

The `setup_plugin_for_backup` command should perform the activities necessary to initialize the remote storage system before backup begins. Set up activities may include creating remote directories, validating connectivity to the remote storage system, checking disks, and so forth.

Arguments

plugin_config_file

The absolute path to the plugin configuration YAML file.

local_backup_dir

The local directory on the Greenplum Database host (master and segments) to which `gpbackup` will write backup files. `gpbackup` creates this local directory.

- When *scope* is `master`, the *local_backup_dir* is the backup directory of the Greenplum Database master.
- When *scope* is `segment`, the *local_backup_dir* is the backup directory of a segment instance. The *contentID* identifies the segment instance.
- When the scope is `segment_host`, the *local_backup_dir* is an arbitrary backup directory on the host.

scope

The execution scope value indicates the host and number of times the plugin command is executed. *scope* can be one of these values:

- `master` - Execute the plugin command once on the master host.
- `segment_host` - Execute the plugin command once on each of the segment hosts.
- `segment` - Execute the plugin command once for each active segment instance on the host running the segment instance. The *contentID* identifies the segment instance.

The Greenplum Database hosts and segment instances are based on the Greenplum Database configuration when the back up was first initiated.

contentID

The contentID of the Greenplum Database master or segment instance corresponding to the scope. *contentID* is passed only when the *scope* is `master` or `segment`.

- When *scope* is `master`, the *contentID* is `-1`.
- When *scope* is `segment`, the *contentID* is the content identifier of an active segment instance.

Exit Code

The `setup_plugin_for_backup` command must exit with a value of 0 on success, non-zero if an error occurs. In the case of a non-zero exit code, `gpbackup` displays the contents of `stderr` to the user.

setup_plugin_for_restore

Plugin command to initialize a storage plugin for the restore operation.

Synopsis

```
plugin_executable setup_plugin_for_restore plugin_config_file local_backup_dir scope
```

```
plugin_executable setup_plugin_for_restore plugin_config_file local_backup_dir scope co
```

Description

`gprestore` invokes the `setup_plugin_for_restore` plugin command during `gprestore` initialization phase. The *scope* argument specifies the execution scope. `gprestore` will invoke the command with each of the *scope* values.

The `setup_plugin_for_restore` command should perform the activities necessary to initialize the remote storage system before a restore operation begins. Set up activities may include creating remote directories, validating connectivity to the remote storage system, etc.

Arguments

plugin_config_file

The absolute path to the plugin configuration YAML file.

local_backup_dir

The local directory on the Greenplum Database host (master and segments) from which `gprestore` reads backup files. `gprestore` creates this local directory.

- When *scope* is `master`, the *local_backup_dir* is the backup directory of the Greenplum Database master.
- When *scope* is `segment`, the *local_backup_dir* is the backup directory of a segment instance. The *contentID* identifies the segment instance.
- When the scope is `segment_host`, the *local_backup_dir* is an arbitrary backup directory on the host.

scope

The execution scope value indicates the host and number of times the plugin command is executed. *scope* can be one of these values:

- `master` - Execute the plugin command once on the master host.
- `segment_host` - Execute the plugin command once on each of the segment hosts.
- `segment` - Execute the plugin command once for each active segment instance on the host running the segment instance. The *contentID* identifies the segment instance.

The Greenplum Database hosts and segment instances are based on the Greenplum Database configuration when the back up was first initiated.

contentID

The contentID of the Greenplum Database master or segment instance corresponding to the scope. *contentID* is passed only when the *scope* is `master` or `segment`.

- When *scope* is `master`, the *contentID* is `-1`.
- When *scope* is `segment`, the *contentID* is the content identifier of an active segment instance.

Exit Code

The `setup_plugin_for_restore` command must exit with a value of 0 on success, non-zero if an error occurs. In the case of a non-zero exit code, `gprestore` displays the contents of `stderr` to the user.

Expanding a Greenplum System

To scale up performance and storage capacity, expand your Greenplum Database system by adding hosts to the system. In general, adding nodes to a Greenplum cluster achieves a linear scaling of performance and storage capacity.

Data warehouses typically grow over time as additional data is gathered and the retention periods increase for existing data. At times, it is necessary to increase database capacity to consolidate different data warehouses into a single database. Additional computing capacity (CPU) may also be needed to accommodate newly added analytics projects. Although it is wise to provide capacity for growth when a system is initially specified, it is not generally possible to invest in resources long before they are required. Therefore, you should expect to execute a database expansion project periodically.

Because of the Greenplum MPP architecture, when you add resources to the system, the capacity and performance are the same as if the system had been originally implemented with the added resources. Unlike data warehouse systems that require substantial downtime in order to dump and restore the data, expanding a Greenplum Database system is a phased process with minimal downtime. Regular and ad hoc workloads can continue while data is redistributed and transactional consistency is maintained. The administrator can schedule the distribution activity to fit into ongoing operations and can pause and resume as needed. Tables can be ranked so that datasets are redistributed in a prioritized sequence, either to ensure that critical workloads benefit from the expanded capacity sooner, or to free disk space needed to redistribute very large tables.

The expansion process uses standard Greenplum Database operations so it is transparent and easy for administrators to troubleshoot. Segment mirroring and any replication mechanisms in place remain active, so fault-tolerance is uncompromised and disaster recovery measures remain effective.

System Expansion Overview

You can perform a Greenplum Database expansion to add segment instances and segment hosts with minimal downtime. In general, adding nodes to a Greenplum cluster achieves a linear scaling of performance and storage capacity.

Data warehouses typically grow over time, often at a continuous pace, as additional data is gathered and the retention period increases for existing data. At times, it is necessary to increase database capacity to consolidate disparate data warehouses into a single database. The data warehouse may also require additional computing capacity (CPU) to accommodate added analytics projects. It is good to provide capacity for growth when a system is initially specified, but even if you anticipate high rates of growth, it is generally unwise to invest in capacity long before it is required. Database expansion, therefore, is a project that you should expect to have to execute periodically.

When you expand your database, you should expect the following qualities:

- Scalable capacity and performance. When you add resources to a Greenplum Database, the capacity and performance are the same as if the system had been originally implemented with the added resources.
- Uninterrupted service during expansion. Regular workloads, both scheduled and ad-hoc, are not interrupted.
- Transactional consistency.
- Fault tolerance. During the expansion, standard fault-tolerance mechanisms—such as segment mirroring—remain active, consistent, and effective.
- Replication and disaster recovery. Any existing replication mechanisms continue to function during expansion. Restore mechanisms needed in case of a failure or catastrophic event remain effective.
- Transparency of process. The expansion process employs standard Greenplum Database mechanisms, so administrators can diagnose and troubleshoot any problems.

- Configurable process. Expansion can be a long running process, but it can be fit into a schedule of ongoing operations. The expansion schema's tables allow administrators to prioritize the order in which tables are redistributed, and the expansion activity can be paused and resumed.

The planning and physical aspects of an expansion project are a greater share of the work than expanding the database itself. It will take a multi-discipline team to plan and execute the project. For on-premise installations, space must be acquired and prepared for the new servers. The servers must be specified, acquired, installed, cabled, configured, and tested. For cloud deployments, similar plans should also be made. *Planning New Hardware Platforms* describes general considerations for deploying new hardware.

After you provision the new hardware platforms and set up their networks, configure the operating systems and run performance tests using Greenplum utilities. The Greenplum Database software distribution includes utilities that are helpful to test and burn-in the new servers before beginning the software phase of the expansion. See *Preparing and Adding Hosts* for steps to prepare the new hosts for Greenplum Database.

Once the new servers are installed and tested, the software phase of the Greenplum Database expansion process begins. The software phase is designed to be minimally disruptive, transactionally consistent, reliable, and flexible.

- The first step of the software phase of expansion process is preparing the Greenplum Database system: adding new segment hosts and initializing new segment instances. This phase can be scheduled to occur during a period of low activity to avoid disrupting ongoing business operations. During the initialization process, the following tasks are performed:
  - Greenplum Database software is installed.
  - Databases and database objects are created in the new segment instances on the new segment hosts.
  - The *gpexpand* schema is created in the postgres database. You can use the tables and view in the schema to monitor and control the expansion process.

  After the system has been updated, the new segment instances on the new segment hosts are available.
  - New segments are immediately available and participate in new queries and data loads. The existing data, however, is skewed. It is concentrated on the original segments and must be redistributed across the new total number of primary segments.
  - Because some of the table data is skewed, some queries might be less efficient because more data motion operations might be needed.
- The last step of the software phase is redistributing table data. Using the expansion control tables in the *gpexpand* schema as a guide, tables are redistributed. For each table:
  - The `gpexand` utility redistributes the table data, across all of the servers, old and new, according to the distribution policy.
  - The table's status is updated in the expansion control tables.
  - After data redistribution, the query optimizer creates more efficient execution plans when data is not skewed.

  When all tables have been redistributed, the expansion is complete.

  Important: The `gprestore` utility cannot restore backups you made before the expansion with the `gpbackup` utility, so back up your databases immediately after the system expansion is complete.

Redistributing table data is a long-running process that creates a large volume of network and disk activity. It can take days to redistribute some very large databases. To minimize the effects of the increased activity on business operations, system administrators can pause and resume expansion activity on an ad hoc basis, or according to a predetermined schedule. Datasets can be prioritized so that critical applications benefit first from the expansion.

In a typical operation, you run the `gpexpand` utility four times with different options during the complete expansion process.

1. To *create an expansion input file*:

   ```
   gpexpand -f hosts_file
   ```

2. To *initialize segments and create the expansion schema*:

   ```
   gpexpand -i input_file
   ```

   `gpexpand` creates a data directory, copies user tables from all existing databases on the new segments, and captures metadata for each table in an expansion schema for status tracking. After this process completes, the expansion operation is committed and irrevocable.

3. *To redistribute table data*:

   ```
   gpexpand -d duration
   ```

   During initialization, `gpexpand` adds and initializes new segment instances. To complete system expansion, you must run `gpexpand` to redistribute data tables across the newly added segment instances. Depending on the size and scale of your system, redistribution can be accomplished in a single session during low-use hours, or you can divide the process into batches over an extended period. Each table or partition is unavailable for read or write operations during redistribution. As each table is redistributed across the new segments, database performance should incrementally improve until it exceeds pre-expansion performance levels.

   You may need to run `gpexpand` several times to complete the expansion in large-scale systems that require multiple redistribution sessions. `gpexpand` can benefit from explicit table redistribution ranking; see *Planning Table Redistribution*.

   Users can access Greenplum Database during initialization, but they may experience performance degradation on systems that rely heavily on hash distribution of tables. Normal operations such as ETL jobs, user queries, and reporting can continue, though users might experience slower response times.

4. To remove the expansion schema:

   ```
   gpexpand -c
   ```

For information about the *gpexpand* utility and the other utilities that are used for system expansion, see the *Greenplum Database Utility Guide*.

Planning Greenplum System Expansion

Careful planning will help to ensure a successful Greenplum expansion project.

The topics in this section help to ensure that you are prepared to execute a system expansion.

- *System Expansion Checklist* is a checklist you can use to prepare for and execute the system expansion process.
- *Planning New Hardware Platforms* covers planning for acquiring and setting up the new hardware.
- *Planning New Segment Initialization* provides information about planning to initialize new segment hosts with `gpexpand`.
- *Planning Table Redistribution* provides information about planning the data redistribution after the new segment hosts have been initialized.

  Important: When expanding a Greenplum Database system, you must disable Greenplum interconnect proxies before adding new hosts and segment instances to the system, and you must update the `gp_interconnect_proxy_addresses` parameter with the newly-added segment instances before you re-enable interconnect proxies. For example, these commands disable Greenplum interconnect proxies by setting the interconnect to the default (`UDPIFC`) and reloading the `postgresql.conf` file to update the Greenplum system configuration.

  ```
  gpconfig -r gp_interconnect_type
  ```

```
gpstop -u
```

For information about Greenplum interconnect proxies, see *Configuring Proxies for the Greenplum Interconnect*.

System Expansion Checklist

This checklist summarizes the tasks for a Greenplum Database system expansion.

Table 24: Greenplum Database System Expansion Checklist

Online Pre-Expansion Tasks * System is up and available	
☐	Devise and execute a plan for ordering, building, and networking new hardware platforms, or provisioning cloud resources.
☐	Devise a database expansion plan. Map the number of segments per host, schedule the downtime period for testing performance and creating the expansion schema, and schedule the intervals for table redistribution.
☐	Perform a complete schema dump.
☐	Install Greenplum Database binaries on new hosts.
☐	Copy SSH keys to the new hosts (`gpssh-exkeys`).
☐	Validate disk I/O and memory bandwidth of the new hardware or cloud resources (`gpcheckperf`).
☐	Validate that the master data directory has no extremely large files in the `pg_log` or `gpperfmon/data` directories.
Offline Pre-Expansion Tasks * The system is unavailable to all user activity during this process.	
☐	Validate that there are no catalog issues (`gpcheckcat`).
☐	Validate disk I/O and memory bandwidth of the combined existing and new hardware or cloud resources (`gpcheckperf`).
Online Segment Instance Initialization * System is up and available	
☐	Prepare an expansion input file (`gpexpand`).

☐	Initialize new segments into the system and create an expansion schema (`gpexpand -i input_file`).
Online Expansion and Table Redistribution * System is up and available	
☐	Before you start table redistribution, stop any automated snapshot processes or other processes that consume disk space.
☐	Redistribute tables through the expanded system (`gpexpand`).
☐	Remove expansion schema (`gpexpand -c`).
☐	**Important:** Run `analyze` to update distribution statistics. During the expansion, use `gpexpand -a`, and post-expansion, use `analyze`.
Back Up Databases * System is up and available	
☐	Back up databases using the `gpbackup` utility. Backups you created before you began the system expansion cannot be restored to the newly expanded system because the `gprestore` utility can only restore backups to a Greenplum Database system with the same number of segments.

Planning New Hardware Platforms

A deliberate, thorough approach to deploying compatible hardware greatly minimizes risk to the expansion process.

Hardware resources and configurations for new segment hosts should match those of the existing hosts. Work with *Pivotal Support* before making a hardware purchase to expand Greenplum Database.

The steps to plan and set up new hardware platforms vary for each deployment. Some considerations include how to:

- Prepare the physical space for the new hardware; consider cooling, power supply, and other physical factors.
- Determine the physical networking and cabling required to connect the new and existing hardware.
- Map the existing IP address spaces and developing a networking plan for the expanded system.
- Capture the system configuration (users, profiles, NICs, and so on) from existing hardware to use as a detailed list for ordering new hardware.
- Create a custom build plan for deploying hardware with the desired configuration in the particular site and environment.

After selecting and adding new hardware to your network environment, ensure you perform the tasks described in *Preparing and Adding Hosts*.

Planning New Segment Initialization

Expanding Greenplum Database can be performed when the system is up and available. Run `gpexpand` to initialize new segment instances into the system and create an expansion schema.

The time required depends on the number of schema objects in the Greenplum system and other factors related to hardware performance. In most environments, the initialization of new segments requires less than thirty minutes offline.

These utilities cannot be run while `gpexpand` is performing segment initialization.

- `gpbackup`
- `gpcheckcat`
- `gpconfig`
- `gppkg`
- `gprestore`

Important: After you begin initializing new segments, you can no longer restore the system using backup files created for the pre-expansion system. When initialization successfully completes, the expansion is committed and cannot be rolled back.

Planning Mirror Segments

If your existing system has mirror segments, the new segments must have mirroring configured. If there are no mirrors configured for existing segments, you cannot add mirrors to new hosts with the `gpexpand` utility. For more information about segment mirroring configurations that are available during system initialization, see *About Segment Mirroring Configurations*.

For Greenplum Database systems with mirror segments, ensure you add enough new host machines to accommodate new mirror segments. The number of new hosts required depends on your mirroring strategy:

- **Group Mirroring** — Add at least two new hosts so the mirrors for the first host can reside on the second host, and the mirrors for the second host can reside on the first. This is the default type of mirroring if you enable segment mirroring during system initialization.
- **Spread Mirroring** — Add at least one more host to the system than the number of segments per host. The number of separate hosts must be greater than the number of segment instances per host to ensure even spreading. You can specify this type of mirroring during system initialization or when you enable segment mirroring for an existing system.
- **Block Mirroring** — Adding one or more blocks of host systems. For example add a block of four or eight hosts. Block mirroring is a custom mirroring configuration. For more information about block mirroring, see *Segment Mirroring Configurations* in the *Greenplum Database Best Practices Guide*.

Increasing Segments Per Host

By default, new hosts are initialized with as many primary segments as existing hosts have. You can increase the segments per host or add new segments to existing hosts.

For example, if existing hosts currently have two segments per host, you can use `gpexpand` to initialize two additional segments on existing hosts for a total of four segments and initialize four new segments on new hosts.

The interactive process for creating an expansion input file prompts for this option; you can also specify new segment directories manually in the input configuration file. For more information, see *Creating an Input File for System Expansion*.

About the Expansion Schema

At initialization, the `gpexpand` utility creates an expansion schema named *gpexpand* in the postgres database.

The expansion schema stores metadata for each table in the system so its status can be tracked throughout the expansion process. The expansion schema consists of two tables and a view for tracking expansion operation progress:

- *gpexpand.status*

- *gpexpand.status_detail*
- *gpexpand.expansion_progress*

Control expansion process aspects by modifying *gpexpand.status_detail*. For example, removing a record from this table prevents the system from expanding the table across new segments. Control the order in which tables are processed for redistribution by updating the `rank` value for a record. For more information, see *Ranking Tables for Redistribution*.

Planning Table Redistribution

Table redistribution is performed while the system is online. For many Greenplum systems, table redistribution completes in a single `gpexpand` session scheduled during a low-use period. Larger systems may require multiple sessions and setting the order of table redistribution to minimize performance impact. Complete the table redistribution in one session if possible.

> **Important:** To perform table redistribution, your segment hosts must have enough disk space to temporarily hold a copy of your largest table. All tables are unavailable for read and write operations during redistribution.

The performance impact of table redistribution depends on the size, storage type, and partitioning design of a table. For any given table, redistributing it with `gpexpand` takes as much time as a `CREATE TABLE AS SELECT` operation would. When redistributing a terabyte-scale fact table, the expansion utility can use much of the available system resources, which could affect query performance or other database workloads.

Managing Redistribution in Large-Scale Greenplum Systems

When planning the redistribution phase, consider the impact of the `ACCESS EXCLUSIVE` lock taken on each table, and the table data redistribution method. User activity on a table can delay its redistribution, but also tables are unavailable for user activity during redistribution.

You can manage the order in which tables are redistributed by adjusting their ranking. See *Ranking Tables for Redistribution*. Manipulating the redistribution order can help adjust for limited disk space and restore optimal query performance for high-priority queries sooner.

Table Redistribution Methods

There are two methods of redistributing data when performing a Greenplum Database expansion.

- `rebuild` - Create a new table, copy all the data from the old to the new table, and replace the old table. This is the default. The rebuild method is similar to creating a new table with a `CREATE TABLE AS SELECT` command. During data redistribution, an `ACCESS EXCLUSIVE` lock is acquired on the table.
- `move` - Scan all the data and perform an `UPDATE` operation to move rows as needed to different segment instances. During data redistribution, an `ACCESS EXCLUSIVE` lock is acquired on the table. In general, this method requires less disk space, however, it creates obsolete table rows and might require a `VACUUM` operation on the table after the data redistribution. Also, this method updates indexes one row at a time, which can be much slower than rebuilding the index with the `CREATE INDEX` command.

Systems with Abundant Free Disk Space

In systems with abundant free disk space (required to store a copy of the largest table), you can focus on restoring optimum query performance as soon as possible by first redistributing important tables that queries use heavily. Assign high ranking to these tables, and schedule redistribution operations for times of low system usage. Run one redistribution process at a time until large or critical tables have been redistributed.

Systems with Limited Free Disk Space

If your existing hosts have limited disk space, you may prefer to first redistribute smaller tables (such as dimension tables) to clear space to store a copy of the largest table. Available disk space on the original segments increases as each table is redistributed across the expanded system. When enough free space exists on all segments to store a copy of the largest table, you can redistribute large or critical tables. Redistribution of large tables requires exclusive locks; schedule this procedure for off-peak hours.

Also consider the following:

- Run multiple parallel redistribution processes during off-peak hours to maximize available system resources.
- When running multiple processes, operate within the connection limits for your Greenplum system. For information about limiting concurrent connections, see *Limiting Concurrent Connections*.

Redistributing Append-Optimized and Compressed Tables

`gpexpand` redistributes append-optimized and compressed append-optimized tables at different rates than heap tables. The CPU capacity required to compress and decompress data tends to increase the impact on system performance. For similar-sized tables with similar data, you may find overall performance differences like the following:

- Uncompressed append-optimized tables expand 10% faster than heap tables.
- Append-optimized tables that are defined to use data compression expand at a significantly slower rate than uncompressed append-optimized tables, potentially up to 80% slower.
- Systems with data compression such as ZFS/LZJB take longer to redistribute.

> **Important:** If your system hosts use data compression, use identical compression settings on the new hosts to avoid disk space shortage.

Redistributing Partitioned Tables

Because the expansion utility can process each individual partition on a large table, an efficient partition design reduces the performance impact of table redistribution. Only the child tables of a partitioned table are set to a random distribution policy. The read/write lock for redistribution applies to only one child table at a time.

Redistributing Indexed Tables

Because the `gpexpand` utility must re-index each indexed table after redistribution, a high level of indexing has a large performance impact. Systems with intensive indexing have significantly slower rates of table redistribution.

Preparing and Adding Hosts

Verify your new host systems are ready for integration into the existing Greenplum system.

To prepare new host systems for expansion, install the Greenplum Database software binaries, exchange the required SSH keys, and run performance tests.

Run performance tests first on the new hosts and then all hosts. Run the tests on all hosts with the system offline so user activity does not distort results.

Generally, you should run performance tests when an administrator modifies host networking or other special conditions in the system. For example, if you will run the expanded system on two network clusters, run tests on each cluster.

> **Note:** Preparing host systems for use by a Greenplum Database system assumes that the new hosts' operating system has been properly configured to match the existing hosts, described in *Configuring Your Systems*.

Adding New Hosts to the Trusted Host Environment

New hosts must exchange SSH keys with the existing hosts to enable Greenplum administrative utilities to connect to all segments without a password prompt. Perform the key exchange process twice with the *gpssh-exkeys* utility.

First perform the process as `root`, for administration convenience, and then as the user `gpadmin`, for management utilities. Perform the following tasks in order:

1. *To exchange SSH keys as root*
2. *To create the gpadmin user*
3. *To exchange SSH keys as the gpadmin user*

> **Note:** The Greenplum Database segment host naming convention is `sdwN` where `sdw` is a prefix and `N` is an integer (`sdw1`, `sdw2` and so on). For hosts with multiple interfaces, the convention is to append a dash (`-`) and number to the host name. For example, `sdw1-1` and `sdw1-2` are the two interface names for host `sdw1`.

To exchange SSH keys as root

1. Create a host file with the existing host names in your array and a separate host file with the new expansion host names. For existing hosts, you can use the same host file used to set up SSH keys in the system. In the files, list all hosts (master, backup master, and segment hosts) with one name per line and no extra lines or spaces. Exchange SSH keys using the configured host names for a given host if you use a multi-NIC configuration. In this example, `mdw` is configured with a single NIC, and `sdw1`, `sdw2`, and `sdw3` are configured with 4 NICs:

```
mdw
sdw1-1
sdw1-2
sdw1-3
sdw1-4
sdw2-1
sdw2-2
sdw2-3
sdw2-4
sdw3-1
sdw3-2
sdw3-3
sdw3-4
```

2. Log in as `root` on the master host, and source the `greenplum_path.sh` file from your Greenplum installation.

```
$ su -
# source /usr/local/greenplum-db/greenplum_path.sh
```

3. Run the `gpssh-exkeys` utility referencing the host list files. For example:

```
# gpssh-exkeys -e /home/gpadmin/existing_hosts_file -x
/home/gpadmin/new_hosts_file
```

4. `gpssh-exkeys` checks the remote hosts and performs the key exchange between all hosts. Enter the `root` user password when prompted. For example:

```
***Enter password for root@hostname: <root_password>
```

To create the gpadmin user

1. Use *gpssh* to create the gpadmin user on all the new segment hosts (if it does not exist already). Use the list of new hosts you created for the key exchange. For example:

```
# gpssh -f new_hosts_file '/usr/sbin/useradd gpadmin -d
/home/gpadmin -s /bin/bash'
```

2. Set a password for the new gpadmin user. On Linux, you can do this on all segment hosts simultaneously using gpssh. For example:

```
# gpssh -f new_hosts_file 'echo gpadmin_password | passwd
gpadmin --stdin'
```

3. Verify the gpadmin user has been created by looking for its home directory:

```
# gpssh -f new_hosts_file ls -l /home
```

To exchange SSH keys as the gpadmin user

1. Log in as gpadmin and run the gpssh-exkeys utility referencing the host list files. For example:

```
# gpssh-exkeys -e /home/gpadmin/existing_hosts_file -x
/home/gpadmin/new_hosts_file
```

2. gpssh-exkeys will check the remote hosts and perform the key exchange between all hosts. Enter the gpadmin user password when prompted. For example:

```
***Enter password for gpadmin@hostname: <gpadmin_password>
```

Validating Disk I/O and Memory Bandwidth

Use the *gpcheckperf* utility to test disk I/O and memory bandwidth.

To run gpcheckperf

1. Run the gpcheckperf utility using the host file for new hosts. Use the -d option to specify the file systems you want to test on each host. You must have write access to these directories. For example:

```
$ gpcheckperf -f new_hosts_file -d /data1 -d /data2 -v
```

2. The utility may take a long time to perform the tests because it is copying very large files between the hosts. When it is finished, you will see the summary results for the Disk Write, Disk Read, and Stream tests.

For a network divided into subnets, repeat this procedure with a separate host file for each subnet.

Integrating New Hardware into the System

Before initializing the system with the new segments, shut down the system with gpstop to prevent user activity from skewing performance test results. Then, repeat the performance tests using host files that include *all* hosts, existing and new.

Initializing New Segments

Use the gpexpand utility to create and initialize the new segment instances and create the expansion schema.

The first time you run *gpexpand* with a valid input file it creates and initializes segment instances and creates the expansion schema. After these steps are completed, running gpexpand detects if the expansion schema has been created and, if so, performs table redistribution.

Creating an Input File for System Expansion

To begin expansion, `gpexpand` requires an input file containing information about the new segments and hosts. If you run `gpexpand` without specifying an input file, the utility displays an interactive interview that collects the required information and automatically creates an input file.

If you create the input file using the interactive interview, you may specify a file with a list of expansion hosts in the interview prompt. If your platform or command shell limits the length of the host list, specifying the hosts with `-f` may be mandatory.

Creating an input file in Interactive Mode

Before you run `gpexpand` to create an input file in interactive mode, ensure you know:

- The number of new hosts (or a hosts file)
- The new hostnames (or a hosts file)
- The mirroring strategy used in existing hosts, if any
- The number of segments to add per host, if any

The utility automatically generates an input file based on this information, `dbid`, `content` ID, and data directory values stored in *gp_segment_configuration*, and saves the file in the current directory.

To create an input file in interactive mode

1. Log in on the master host as the user who will run your Greenplum Database system; for example, `gpadmin`.
2. Run `gpexpand`. The utility displays messages about how to prepare for an expansion operation, and it prompts you to quit or continue.

   Optionally, specify a hosts file using `-f`. For example:

   ```
   $ gpexpand -f /home/gpadmin/new_hosts_file
   ```
3. At the prompt, select `Y` to continue.
4. Unless you specified a hosts file using `-f`, you are prompted to enter hostnames. Enter a comma separated list of the hostnames of the new expansion hosts. Do not include interface hostnames. For example:

   ```
   > sdw4, sdw5, sdw6, sdw7
   ```

   To add segments to existing hosts only, enter a blank line at this prompt. Do not specify `localhost` or any existing host name.
5. Enter the mirroring strategy used in your system, if any. Options are `spread|grouped|none`. The default setting is `grouped`.

   Ensure you have enough hosts for the selected grouping strategy. For more information about mirroring, see *Planning Mirror Segments*.
6. Enter the number of new primary segments to add, if any. By default, new hosts are initialized with the same number of primary segments as existing hosts. Increase segments per host by entering a number greater than zero. The number you enter will be the number of additional segments initialized on all hosts. For example, if existing hosts currently have two segments each, entering a value of 2 initializes two more segments on existing hosts, and four segments on new hosts.
7. If you are adding new primary segments, enter the new primary data directory root for the new segments. Do not specify the actual data directory name, which is created automatically by `gpexpand` based on the existing data directory names.

For example, if your existing data directories are as follows:

```
/gpdata/primary/gp0
/gpdata/primary/gp1
```

then enter the following (one at each prompt) to specify the data directories for two new primary segments:

```
/gpdata/primary
/gpdata/primary
```

When the initialization runs, the utility creates the new directories `gp2` and `gp3` under `/gpdata/primary`.

8. If you are adding new mirror segments, enter the new mirror data directory root for the new segments. Do not specify the data directory name; it is created automatically by `gpexpand` based on the existing data directory names.

   For example, if your existing data directories are as follows:

   ```
   /gpdata/mirror/gp0
   /gpdata/mirror/gp1
   ```

   enter the following (one at each prompt) to specify the data directories for two new mirror segments:

   ```
   /gpdata/mirror
   /gpdata/mirror
   ```

   When the initialization runs, the utility will create the new directories `gp2` and `gp3` under `/gpdata/mirror`.

   These primary and mirror root directories for new segments must exist on the hosts, and the user running `gpexpand` must have permissions to create directories in them.

   After you have entered all required information, the utility generates an input file and saves it in the current directory. For example:

   ```
   gpexpand_inputfile_yyyymmdd_145134
   ```

Expansion Input File Format

Use the interactive interview process to create your own input file unless your expansion scenario has atypical needs.

The format for expansion input files is:

```
hostname|address|port|datadir|dbid|content|preferred_role
```

For example:

```
sdw5|sdw5-1|50011|/gpdata/primary/gp9|11|9|p
sdw5|sdw5-2|50012|/gpdata/primary/gp10|12|10|p
sdw5|sdw5-2|60011|/gpdata/mirror/gp9|13|9|m
sdw5|sdw5-1|60012|/gpdata/mirror/gp10|14|10|m
```

For each new segment, this format of expansion input file requires the following:

Table 25: Data for the expansion configuration file

Parameter	Valid Values	Description
hostname	Hostname	Hostname for the segment host.
port	An available port number	Database listener port for the segment, incremented on the existing segment *port* base number.
datadir	Directory name	The data directory location for a segment as per the gp_segment_ configuration system catalog.
dbid	Integer. Must not conflict with existing *dbid* values.	Database ID for the segment. The values you enter should be incremented sequentially from existing *dbid* values shown in the system catalog `gp_segment_ configuration`. For example, to add four segment instances to an existing ten-segment array with *dbid* values of 1-10, list new *dbid* values of 11, 12, 13 and 14.
content	Integer. Must not conflict with existing *content* values.	The content ID of the segment. A primary segment and its mirror should have the same content ID, incremented sequentially from existing values. For more information, see *content* in the reference for `gp_segment_ configuration`.
preferred_role	`p | m`	Determines whether this segment is a primary or mirror. Specify `p` for primary and `m` for mirror.

Running gpexpand to Initialize New Segments

After you have created an input file, run `gpexpand` to initialize new segment instances.

To run gpexpand with an input file

1. Log in on the master host as the user who will run your Greenplum Database system; for example, `gpadmin`.
2. Run the `gpexpand` utility, specifying the input file with `-i`. For example:

   ```
   $ gpexpand -i input_file
   ```

   The utility detects if an expansion schema exists for the Greenplum Database system. If a *gpexpand* schema exists, remove it with `gpexpand -c` before you start a new expansion operation. See *Removing the Expansion Schema*.

   When the new segments are initialized and the expansion schema is created, the utility prints a success message and exits.

When the initialization process completes, you can connect to Greenplum Database and view the expansion schema. The *gpexpand* schema resides in the postgres database. For more information, see *About the Expansion Schema*.

After segment initialization is complete, *redistribute the tables* to balance existing data over the new segments.

Monitoring the Cluster Expansion State

At any time, you can check the state of cluster expansion by running the `gpstate` utility with the `-x` flag:

```
$ gpstate -x
```

If the expansion schema exists in the postgres database, `gpstate -x` reports on the progress of the expansion. During the first expansion phase, `gpstate` reports on the progress of new segment initialization. During the second phase, `gpstate` reports on the progress of table redistribution, and whether redistribution is paused or active.

You can also query the expansion schema to see expansion status. See *Monitoring Table Redistribution* for more information.

Rolling Back a Failed Expansion Setup

You can roll back an expansion setup operation (adding segment instances and segment hosts) only if the operation fails.

If the expansion fails during the initialization step, while the database is down, you must first restart the database in master-only mode by running the `gpstart -m` command.

Roll back the failed expansion with the following command:

```
gpexpand --rollback
```

Redistributing Tables

Redistribute tables to balance existing data over the newly expanded cluster.

After creating an expansion schema, you can redistribute tables across the entire system with *`gpexpand`*. Plan to run this during low-use hours when the utility's CPU usage and table locks have minimal impact on operations. Rank tables to redistribute the largest or most critical tables first.

> **Note:** When redistributing data, Greenplum Database must be running in production mode. Greenplum Database cannot be in restricted mode or in master mode. The *`gpstart`* options `-R` or `-m` cannot be specified to start Greenplum Database.

While table redistribution is underway, any new tables or partitions created are distributed across all segments exactly as they would be under normal operating conditions. Queries can access all segments, even before the relevant data is redistributed to tables on the new segments. The table or partition being redistributed is locked and unavailable for read or write operations. When its redistribution completes, normal operations resume.

- *Ranking Tables for Redistribution*
- *Redistributing Tables Using gpexpand*
- *Monitoring Table Redistribution*

Ranking Tables for Redistribution

For large systems, you can control the table redistribution order. Adjust tables' `rank` values in the expansion schema to prioritize heavily-used tables and minimize performance impact. Available free disk space can affect table ranking; see *Managing Redistribution in Large-Scale Greenplum Systems*.

To rank tables for redistribution by updating `rank` values in *gpexpand.status_detail*, connect to Greenplum Database using `psql` or another supported client. Update *gpexpand.status_detail* with commands such as:

```
=> UPDATE gpexpand.status_detail SET rank=10;

=> UPDATE gpexpand.status_detail SET rank=1 WHERE fq_name =
 'public.lineitem';
=> UPDATE gpexpand.status_detail SET rank=2 WHERE fq_name = 'public.orders';
```

These commands lower the priority of all tables to `10` and then assign a rank of `1` to `lineitem` and a rank of `2` to `orders`. When table redistribution begins, `lineitem` is redistributed first, followed by `orders` and all other tables in *gpexpand.status_detail*. To exclude a table from redistribution, remove the table from the *gpexpand.status_detail* table.

Redistributing Tables Using gpexpand

To redistribute tables with gpexpand

1. Log in on the master host as the user who will run your Greenplum Database system, for example, `gpadmin`.
2. Run the `gpexpand` utility. You can use the `-d` or `-e` option to define the expansion session time period. For example, to run the utility for up to 60 consecutive hours:

   ```
   $ gpexpand -d 60:00:00
   ```

   The utility redistributes tables until the last table in the schema completes or it reaches the specified duration or end time. `gpexpand` updates the status and time in *gpexpand.status* when a session starts and finishes.

   > **Note:** After completing table redistribution, run the `VACUUM ANALYZE` and `REINDEX`commands on the catalog tables to update table statistics, and rebuild indexes. See *Routine Vacuum and Analyze* in the *Administration Guide* and *VACUUM* in the *Reference Guide*.

Monitoring Table Redistribution

During the table redistribution process you can query the expansion schema to view:

- a current progress summary, the estimated rate of table redistribution, and the estimated time to completion. Use *gpexpand.expansion_progress*, as described in *Viewing Expansion Status*.
- per-table status information, using *gpexpand.status_detail*. See *Viewing Table Status*.

See also *Monitoring the Cluster Expansion State* for information about monitoring the overall expansion progress with the `gpstate` utility.

Viewing Expansion Status

After the first table completes redistribution, *gpexpand.expansion_progress* calculates its estimates and refreshes them based on all tables' redistribution rates. Calculations restart each time you start a table redistribution session with `gpexpand`. To monitor progress, connect to Greenplum Database using `psql` or another supported client; query *gpexpand.expansion_progress* with a command like the following:

```
=# SELECT * FROM gpexpand.expansion_progress;
             name             |         value
------------------------------+----------------------
 Bytes Left                   | 5534842880
 Bytes Done                   | 142475264
 Estimated Expansion Rate     | 680.75667095996092 MB/s
 Estimated Time to Completion | 00:01:01.008047
 Tables Expanded              | 4
```

```
 Tables Left                  | 4
(6 rows)
```

Viewing Table Status

The table *gpexpand.status_detail* stores status, time of last update, and more facts about each table in the schema. To see a table's status, connect to Greenplum Database using `psql` or another supported client and query *gpexpand.status_detail*:

```
=> SELECT status, expansion_started, source_bytes FROM
gpexpand.status_detail WHERE fq_name = 'public.sales';
  status   |     expansion_started      | source_bytes
-----------+----------------------------+--------------
 COMPLETED | 2017-02-20 10:54:10.043869 |   4929748992
(1 row)
```

Removing the Expansion Schema

To clean up after expanding the Greenplum cluster, remove the expansion schema.

You can safely remove the expansion schema after the expansion operation is complete and verified. To run another expansion operation on a Greenplum system, first remove the existing expansion schema.

To remove the expansion schema

1. Log in on the master host as the user who will be running your Greenplum Database system (for example, `gpadmin`).
2. Run the `gpexpand` utility with the `-c` option. For example:

```
$ gpexpand -c
```

Note: Some systems require you to press Enter twice.

Migrating Data with gpcopy

You can use the `gpcopy` utility to transfer data between databases in different Greenplum Database clusters.

gpcopy is a high-performance utility that can copy metadata and data from one Greenplum database to another Greenplum database. You can migrate the entire contents of a database, or just selected tables. The clusters can have different Greenplum Database versions. For example, you can use `gpcopy` to migrate data from a Greenplum Database version 4.3.26 (or later) system to a 5.9 (or later) or a 6.x Greenplum system, or from a Greenplum Database version 5.9+ system to a Greenplum 6.x system.

Note: The `gpcopy` utility is available as a separate download for the commercial release of Pivotal Greenplum Database. See the *Pivotal gpcopy Documentation*.

Monitoring a Greenplum System

You can monitor a Greenplum Database system using a variety of tools included with the system or available as add-ons.

Observing the Greenplum Database system day-to-day performance helps administrators understand the system behavior, plan workflow, and troubleshoot problems. This chapter discusses tools for monitoring database performance and activity.

Also, be sure to review *Recommended Monitoring and Maintenance Tasks* for monitoring activities you can script to quickly detect problems in the system.

Monitoring Database Activity and Performance

Greenplum Database includes an optional system monitoring and management database, `gpperfmon`, that administrators can enable. The `gpperfmon_install` command-line utility creates the `gpperfmon` database and enables data collection agents that collect and store query and system metrics in the database. Administrators can query metrics in the `gpperfmon` database. See the documentation for the `gpperfmon` database in the *Greenplum Database Reference Guide*.

Pivotal Greenplum Command Center, an optional web-based interface, provides cluster status information, graphical administrative tools, real-time query monitoring, and historical cluster and query data. Download the Greenplum Command Center package from *Pivotal Network* and view the documentation at the *Greenplum Command Center Documentation* web site.

Monitoring System State

As a Greenplum Database administrator, you must monitor the system for problem events such as a segment going down or running out of disk space on a segment host. The following topics describe how to monitor the health of a Greenplum Database system and examine certain state information for a Greenplum Database system.

- *Checking System State*
- *Checking Disk Space Usage*
- *Checking for Data Distribution Skew*
- *Viewing Metadata Information about Database Objects*
- *Viewing Session Memory Usage Information*
- *Viewing Query Workfile Usage Information*

Checking System State

A Greenplum Database system is comprised of multiple PostgreSQL instances (the master and segments) spanning multiple machines. To monitor a Greenplum Database system, you need to know information about the system as a whole, as well as status information of the individual instances. The `gpstate` utility provides status information about a Greenplum Database system.

Viewing Master and Segment Status and Configuration

The default `gpstate` action is to check segment instances and show a brief status of the valid and failed segments. For example, to see a quick status of your Greenplum Database system:

```
$ gpstate
```

To see more detailed information about your Greenplum Database array configuration, use `gpstate` with the `-s` option:

```
$ gpstate -s
```

Viewing Your Mirroring Configuration and Status

If you are using mirroring for data redundancy, you may want to see the list of mirror segment instances in the system, their current synchronization status, and the mirror to primary mapping. For example, to see the mirror segments in the system and their status:

```
$ gpstate -m
```

To see the primary to mirror segment mappings:

```
$ gpstate -c
```

To see the status of the standby master mirror:

```
$ gpstate -f
```

Checking Disk Space Usage

A database administrator's most important monitoring task is to make sure the file systems where the master and segment data directories reside do not grow to more than 70 percent full. A filled data disk will not result in data corruption, but it may prevent normal database activity from continuing. If the disk grows too full, it can cause the database server to shut down.

You can use the `gp_disk_free` external table in the `gp_toolkit` administrative schema to check for remaining free space (in kilobytes) on the segment host file systems. For example:

```
=# SELECT * FROM gp_toolkit.gp_disk_free
   ORDER BY dfsegment;
```

Checking Sizing of Distributed Databases and Tables

The `gp_toolkit` administrative schema contains several views that you can use to determine the disk space usage for a distributed Greenplum Database database, schema, table, or index.

For a list of the available sizing views for checking database object sizes and disk space, see the *Greenplum Database Reference Guide*.

Viewing Disk Space Usage for a Database

To see the total size of a database (in bytes), use the *gp_size_of_database* view in the *gp_toolkit* administrative schema. For example:

```
=> SELECT * FROM gp_toolkit.gp_size_of_database
   ORDER BY sodddatname;
```

Viewing Disk Space Usage for a Table

The *gp_toolkit* administrative schema contains several views for checking the size of a table. The table sizing views list the table by object ID (not by name). To check the size of a table by name, you must look up the relation name (`relname`) in the *pg_class* table. For example:

```
=> SELECT relname AS name, sotdsize AS size, sotdtoastsize
   AS toast, sotdadditionalsize AS other
   FROM gp_toolkit.gp_size_of_table_disk as sotd, pg_class
   WHERE sotd.sotdoid=pg_class.oid ORDER BY relname;
```

For a list of the available table sizing views, see the *Greenplum Database Reference Guide*.

Viewing Disk Space Usage for Indexes

The *gp_toolkit* administrative schema contains a number of views for checking index sizes. To see the total size of all index(es) on a table, use the *gp_size_of_all_table_indexes* view. To see the size of a particular index, use the *gp_size_of_index* view. The index sizing views list tables and indexes by object ID (not

by name). To check the size of an index by name, you must look up the relation name (`relname`) in the *pg_class* table. For example:

```
=> SELECT soisize, relname as indexname
   FROM pg_class, gp_toolkit.gp_size_of_index
   WHERE pg_class.oid=gp_size_of_index.soioid
   AND pg_class.relkind='i';
```

Checking for Data Distribution Skew

All tables in Greenplum Database are distributed, meaning their data is divided across all of the segments in the system. Unevenly distributed data may diminish query processing performance. A table's distribution policy, set at table creation time, determines how the table's rows are distributed. For information about choosing the table distribution policy, see the following topics:

- *Viewing a Table's Distribution Key*
- *Viewing Data Distribution*
- *Checking for Query Processing Skew*

The *gp_toolkit* administrative schema also contains a number of views for checking data distribution skew on a table. For information about how to check for uneven data distribution, see the *Greenplum Database Reference Guide*.

Viewing a Table's Distribution Key

To see the columns used as the data distribution key for a table, you can use the `\d+` meta-command in `psql` to examine the definition of a table. For example:

```
=# \d+ sales
                Table "retail.sales"
 Column      |     Type     | Modifiers | Description
-------------+--------------+-----------+-------------
 sale_id     | integer      |           |
 amt         | float        |           |
 date        | date         |           |
Has OIDs: no
Distributed by: (sale_id)
```

When you create a *replicated* table, Greenplum Database stores all rows in the table on every segment. Replicated tables have no distribution key. Where the `\d+` meta-command reports the distribution key for a normally distributed table, it shows `Distributed Replicated` for a replicated table.

Viewing Data Distribution

To see the data distribution of a table's rows (the number of rows on each segment), you can run a query such as:

```
=# SELECT gp_segment_id, count(*)
   FROM table_name GROUP BY gp_segment_id;
```

A table is considered to have a balanced distribution if all segments have roughly the same number of rows.

> **Note:** If you run this query on a replicated table, it fails because Greenplum Database does not permit user queries to reference the system column `gp_segment_id` (or the system columns `ctid`, `cmin`, `cmax`, `xmin`, and `xmax`) in replicated tables. Because every segment has all of the tables' rows, replicated tables are evenly distributed by definition.

Checking for Query Processing Skew

When a query is being processed, all segments should have equal workloads to ensure the best possible performance. If you identify a poorly-performing query, you may need to investigate further using the `EXPLAIN` command. For information about using the `EXPLAIN` command and query profiling, see *Query Profiling*.

Query processing workload can be skewed if the table's data distribution policy and the query predicates are not well matched. To check for processing skew, you can run a query such as:

```
=# SELECT gp_segment_id, count(*) FROM table_name
   WHERE column='value' GROUP BY gp_segment_id;
```

This will show the number of rows returned by segment for the given `WHERE` predicate.

As noted in *Viewing Data Distribution*, this query will fail if you run it on a replicated table because you cannot reference the `gp_segment_id` system column in a query on a replicated table.

Avoiding an Extreme Skew Warning

You may receive the following warning message while executing a query that performs a hash join operation:

```
Extreme skew in the innerside of Hashjoin
```

This occurs when the input to a hash join operator is skewed. It does not prevent the query from completing successfully. You can follow these steps to avoid skew in the plan:

1. Ensure that all fact tables are analyzed.
2. Verify that any populated temporary table used by the query is analyzed.
3. View the `EXPLAIN ANALYZE` plan for the query and look for the following:
   - If there are scans with multi-column filters that are producing more rows than estimated, then set the `gp_selectivity_damping_factor` server configuration parameter to 2 or higher and retest the query.
   - If the skew occurs while joining a single fact table that is relatively small (less than 5000 rows), set the `gp_segments_for_planner` server configuration parameter to 1 and retest the query.
4. Check whether the filters applied in the query match distribution keys of the base tables. If the filters and distribution keys are the same, consider redistributing some of the base tables with different distribution keys.
5. Check the cardinality of the join keys. If they have low cardinality, try to rewrite the query with different joining columns or or additional filters on the tables to reduce the number of rows. These changes could change the query semantics.

Viewing Metadata Information about Database Objects

Greenplum Database tracks various metadata information in its system catalogs about the objects stored in a database, such as tables, views, indexes and so on, as well as global objects such as roles and tablespaces.

Viewing the Last Operation Performed

You can use the system views *pg_stat_operations* and *pg_stat_partition_operations* to look up actions performed on an object, such as a table. For example, to see the actions performed on a table, such as when it was created and when it was last vacuumed and analyzed:

```
=> SELECT schemaname as schema, objname as table,
   usename as role, actionname as action,
   subtype as type, statime as time
```

```
   FROM pg_stat_operations
   WHERE objname='cust';
 schema | table | role | action  | type  | time
--------+-------+------+---------+-------+--------------------------
  sales | cust  | main | CREATE  | TABLE | 2016-02-09 18:10:07.867977-08
  sales | cust  | main | VACUUM  |       | 2016-02-10 13:32:39.068219-08
  sales | cust  | main | ANALYZE |       | 2016-02-25 16:07:01.157168-08
(3 rows)
```

Viewing the Definition of an Object

To see the definition of an object, such as a table or view, you can use the `\d+` meta-command when working in `psql`. For example, to see the definition of a table:

```
=> \d+ mytable
```

Viewing Session Memory Usage Information

You can create and use the *session_level_memory_consumption* view that provides information about the current memory utilization for sessions that are running queries on Greenplum Database. The view contains session information and information such as the database that the session is connected to, the query that the session is currently running, and memory consumed by the session processes.

Creating the session_level_memory_consumption View

To create the *session_state.session_level_memory_consumption* view in a Greenplum Database, run the script `CREATE EXTENSION gp_internal_tools;` once for each database. For example, to install the view in the database `testdb`, use this command:

```
$ psql -d testdb -c "CREATE EXTENSION gp_internal_tools;"
```

The session_level_memory_consumption View

The *session_state.session_level_memory_consumption* view provides information about memory consumption and idle time for sessions that are running SQL queries.

When resource queue-based resource management is active, the column `is_runaway` indicates whether Greenplum Database considers the session a runaway session based on the vmem memory consumption of the session's queries. Under the resource queue-based resource management scheme, Greenplum Database considers the session a runaway when the queries consume an excessive amount of memory. The Greenplum Database server configuration parameter `runaway_detector_activation_percent` governs the conditions under which Greenplum Database considers a session a runaway session.

The `is_runaway`, `runaway_vmem_mb`, and `runaway_command_cnt` columns are not applicable when resource group-based resource management is active.

Table 26: session_state.session_level_memory_consumption

column	type	references	description
`datname`	name		Name of the database that the session is connected to.
`sess_id`	integer		Session ID.
`usename`	name		Name of the session user.

column	type	references	description
query	text		Current SQL query that the session is running.
segid	integer		Segment ID.
vmem_mb	integer		Total vmem memory usage for the session in MB.
is_runaway	boolean		Session is marked as runaway on the segment.
qe_count	integer		Number of query processes for the session.
active_qe_count	integer		Number of active query processes for the session.
dirty_qe_count	integer		Number of query processes that have not yet released their memory. The value is -1 for sessions that are not running.
runaway_vmem_mb	integer		Amount of vmem memory that the session was consuming when it was marked as a runaway session.
runaway_command_cnt	integer		Command count for the session when it was marked as a runaway session.
idle_start	timestamptz		The last time a query process in this session became idle.

Viewing Query Workfile Usage Information

The Greenplum Database administrative schema *gp_toolkit* contains views that display information about Greenplum Database workfiles. Greenplum Database creates workfiles on disk if it does not have sufficient memory to execute the query in memory. This information can be used for troubleshooting and tuning queries. The information in the views can also be used to specify the values for the Greenplum Database configuration parameters `gp_workfile_limit_per_query` and `gp_workfile_limit_per_segment`.

These are the views in the schema *gp_toolkit*:

- The gp_workfile_entries view contains one row for each operator using disk space for workfiles on a segment at the current time.

- The gp_workfile_usage_per_query view contains one row for each query using disk space for workfiles on a segment at the current time.
- The gp_workfile_usage_per_segment view contains one row for each segment. Each row displays the total amount of disk space used for workfiles on the segment at the current time.

For information about using *gp_toolkit*, see *Using gp_toolkit*.

Viewing the Database Server Log Files

Every database instance in Greenplum Database (master and segments) runs a PostgreSQL database server with its own server log file. Log files are created in the `pg_log` directory of the master and each segment data directory.

Log File Format

The server log files are written in comma-separated values (CSV) format. Some log entries will not have values for all log fields. For example, only log entries associated with a query worker process will have the `slice_id` populated. You can identify related log entries of a particular query by the query's session identifier (`gp_session_id`) and command identifier (`gp_command_count`).

The following fields are written to the log:

Table 27: Greenplum Database Server Log Format

#	Field Name	Data Type	Description
1	event_time	timestamp with time zone	Time that the log entry was written to the log
2	user_name	varchar(100)	The database user name
3	database_name	varchar(100)	The database name
4	process_id	varchar(10)	The system process ID (prefixed with "p")
5	thread_id	varchar(50)	The thread count (prefixed with "th")
6	remote_host	varchar(100)	On the master, the hostname/address of the client machine. On the segment, the hostname/address of the master.
7	remote_port	varchar(10)	The segment or master port number
8	session_start_time	timestamp with time zone	Time session connection was opened
9	transaction_id	int	Top-level transaction ID on the master. This ID is the parent of any subtransactions.
10	gp_session_id	text	Session identifier number (prefixed with "con")
11	gp_command_ count	text	The command number within a session (prefixed with "cmd")
12	gp_segment	text	The segment content identifier (prefixed with "seg" for primaries or "mir" for mirrors). The master always has a content ID of -1.
13	slice_id	text	The slice ID (portion of the query plan being executed)
14	distr_tranx_id	text	Distributed transaction ID
15	local_tranx_id	text	Local transaction ID

#	Field Name	Data Type	Description
16	sub_tranx_id	text	Subtransaction ID
17	event_severity	varchar(10)	Values include: LOG, ERROR, FATAL, PANIC, DEBUG1, DEBUG2
18	sql_state_code	varchar(10)	SQL state code associated with the log message
19	event_message	text	Log or error message text
20	event_detail	text	Detail message text associated with an error or warning message
21	event_hint	text	Hint message text associated with an error or warning message
22	internal_query	text	The internally-generated query text
23	internal_query_pos	int	The cursor index into the internally-generated query text
24	event_context	text	The context in which this message gets generated
25	debug_query_ string	text	User-supplied query string with full detail for debugging. This string can be modified for internal use.
26	error_cursor_pos	int	The cursor index into the query string
27	func_name	text	The function in which this message is generated
28	file_name	text	The internal code file where the message originated
29	file_line	int	The line of the code file where the message originated
30	stack_trace	text	Stack trace text associated with this message

Searching the Greenplum Server Log Files

Greenplum Database provides a utility called `gplogfilter` can search through a Greenplum Database log file for entries matching the specified criteria. By default, this utility searches through the Greenplum Database master log file in the default logging location. For example, to display the last three lines of the master log file:

```
$ gplogfilter -n 3
```

To search through all segment log files simultaneously, run `gplogfilter` through the `gpssh` utility. For example, to display the last three lines of each segment log file:

```
$ gpssh -f seg_host_file
```

```
=> source /usr/local/greenplum-db/greenplum_path.sh
=> gplogfilter -n 3 /gpdata/gp*/pg_log/gpdb*.log
```

Using gp_toolkit

Use the Greenplum Database administrative schema *gp_toolkit* to query the system catalogs, log files, and operating environment for system status information. The `gp_toolkit` schema contains several views you can access using SQL commands. The *gp_toolkit* schema is accessible to all database users.

Some objects require superuser permissions. Use a command similar to the following to add the *gp_toolkit* schema to your schema search path:

```
=> ALTER ROLE myrole SET search_path TO myschema,gp_toolkit;
```

For a description of the available administrative schema views and their usages, see the *Greenplum Database Reference Guide*.

SQL Standard Error Codes

The following table lists all the defined error codes. Some are not used, but are defined by the SQL standard. The error classes are also shown. For each error class there is a standard error code having the last three characters 000. This code is used only for error conditions that fall within the class but do not have any more-specific code assigned.

The PL/pgSQL condition name for each error code is the same as the phrase shown in the table, with underscores substituted for spaces. For example, code 22012, DIVISION BY ZERO, has condition name DIVISION_BY_ZERO. Condition names can be written in either upper or lower case.

> **Note:** PL/pgSQL does not recognize warning, as opposed to error, condition names; those are classes 00, 01, and 02.

Table 28: SQL Codes

Error Code	Meaning	Constant
Class 00 — Successful Completion		
00000	SUCCESSFUL COMPLETION	successful_completion
Class 01 — Warning		
01000	WARNING	warning
0100C	DYNAMIC RESULT SETS RETURNED	dynamic_result_sets_returned
01008	IMPLICIT ZERO BIT PADDING	implicit_zero_bit_padding
01003	NULL VALUE ELIMINATED IN SET FUNCTION	null_value_eliminated_in_set_function
01007	PRIVILEGE NOT GRANTED	privilege_not_granted
01006	PRIVILEGE NOT REVOKED	privilege_not_revoked
01004	STRING DATA RIGHT TRUNCATION	string_data_right_truncation
01P01	DEPRECATED FEATURE	deprecated_feature
Class 02 — No Data (this is also a warning class per the SQL standard)		
02000	NO DATA	no_data
02001	NO ADDITIONAL DYNAMIC RESULT SETS RETURNED	no_additional_dynamic_result_sets_returned
Class 03 — SQL Statement Not Yet Complete		
03000	SQL STATEMENT NOT YET COMPLETE	sql_statement_not_yet_complete
Class 08 — Connection Exception		
08000	CONNECTION EXCEPTION	connection_exception

Error Code	Meaning	Constant
08003	CONNECTION DOES NOT EXIST	connection_does_not_exist
08006	CONNECTION FAILURE	connection_failure
08001	SQLCLIENT UNABLE TO ESTABLISH SQLCONNECTION	sqlclient_unable_to_establish_sqlconnection
08004	SQLSERVER REJECTED ESTABLISHMENT OF SQLCONNECTION	sqlserver_rejected_establishment_of_ sqlconnection
08007	TRANSACTION RESOLUTION UNKNOWN	transaction_resolution_unknown
08P01	PROTOCOL VIOLATION	protocol_violation
Class 09 — Triggered Action Exception		
09000	TRIGGERED ACTION EXCEPTION	triggered_action_exception
Class 0A — Feature Not Supported		
0A000	FEATURE NOT SUPPORTED	feature_not_supported
Class 0B — Invalid Transaction Initiation		
0B000	INVALID TRANSACTION INITIATION	invalid_transaction_initiation
Class 0F — Locator Exception		
0F000	LOCATOR EXCEPTION	locator_exception
0F001	INVALID LOCATOR SPECIFICATION	invalid_locator_specification
Class 0L — Invalid Grantor		
0L000	INVALID GRANTOR	invalid_grantor
0LP01	INVALID GRANT OPERATION	invalid_grant_operation
Class 0P — Invalid Role Specification		
0P000	INVALID ROLE SPECIFICATION	invalid_role_specification
Class 21 — Cardinality Violation		
21000	CARDINALITY VIOLATION	cardinality_violation
Class 22 — Data Exception		
22000	DATA EXCEPTION	data_exception
2202E	ARRAY SUBSCRIPT ERROR	array_subscript_error
22021	CHARACTER NOT IN REPERTOIRE	character_not_in_repertoire
22008	DATETIME FIELD OVERFLOW	datetime_field_overflow
22012	DIVISION BY ZERO	division_by_zero
22005	ERROR IN ASSIGNMENT	error_in_assignment
2200B	ESCAPE CHARACTER CONFLICT	escape_character_conflict
22022	INDICATOR OVERFLOW	indicator_overflow
22015	INTERVAL FIELD OVERFLOW	interval_field_overflow

Error Code	Meaning	Constant
2201E	INVALID ARGUMENT FOR LOGARITHM	invalid_argument_for_logarithm
2201F	INVALID ARGUMENT FOR POWER FUNCTION	invalid_argument_for_power_function
2201G	INVALID ARGUMENT FOR WIDTH BUCKET FUNCTION	invalid_argument_for_width_bucket_function
22018	INVALID CHARACTER VALUE FOR CAST	invalid_character_value_for_cast
22007	INVALID DATETIME FORMAT	invalid_datetime_format
22019	INVALID ESCAPE CHARACTER	invalid_escape_character
2200D	INVALID ESCAPE OCTET	invalid_escape_octet
22025	INVALID ESCAPE SEQUENCE	invalid_escape_sequence
22P06	NONSTANDARD USE OF ESCAPE CHARACTER	nonstandard_use_of_escape_character
22010	INVALID INDICATOR PARAMETER VALUE	invalid_indicator_parameter_value
22020	INVALID LIMIT VALUE	invalid_limit_value
22023	INVALID PARAMETER VALUE	invalid_parameter_value
2201B	INVALID REGULAR EXPRESSION	invalid_regular_expression
22009	INVALID TIME ZONE DISPLACEMENT VALUE	invalid_time_zone_displacement_value
2200C	INVALID USE OF ESCAPE CHARACTER	invalid_use_of_escape_character
2200G	MOST SPECIFIC TYPE MISMATCH	most_specific_type_mismatch
22004	NULL VALUE NOT ALLOWED	null_value_not_allowed
22002	NULL VALUE NO INDICATOR PARAMETER	null_value_no_indicator_parameter
22003	NUMERIC VALUE OUT OF RANGE	numeric_value_out_of_range
22026	STRING DATA LENGTH MISMATCH	string_data_length_mismatch
22001	STRING DATA RIGHT TRUNCATION	string_data_right_truncation
22011	SUBSTRING ERROR	substring_error
22027	TRIM ERROR	trim_error
22024	UNTERMINATED C STRING	unterminated_c_string
2200F	ZERO LENGTH CHARACTER STRING	zero_length_character_string
22P01	FLOATING POINT EXCEPTION	floating_point_exception
22P02	INVALID TEXT REPRESENTATION	invalid_text_representation
22P03	INVALID BINARY REPRESENTATION	invalid_binary_representation
22P04	BAD COPY FILE FORMAT	bad_copy_file_format

Error Code	Meaning	Constant
22P05	UNTRANSLATABLE CHARACTER	untranslatable_character
Class 23 — Integrity Constraint Violation		
23000	INTEGRITY CONSTRAINT VIOLATION	integrity_constraint_violation
23001	RESTRICT VIOLATION	restrict_violation
23502	NOT NULL VIOLATION	not_null_violation
23503	FOREIGN KEY VIOLATION	foreign_key_violation
23505	UNIQUE VIOLATION	unique_violation
23514	CHECK VIOLATION	check_violation
Class 24 — Invalid Cursor State		
24000	INVALID CURSOR STATE	invalid_cursor_state
Class 25 — Invalid Transaction State		
25000	INVALID TRANSACTION STATE	invalid_transaction_state
25001	ACTIVE SQL TRANSACTION	active_sql_transaction
25002	BRANCH TRANSACTION ALREADY ACTIVE	branch_transaction_already_active
25008	HELD CURSOR REQUIRES SAME ISOLATION LEVEL	held_cursor_requires_same_isolation_level
25003	INAPPROPRIATE ACCESS MODE FOR BRANCH TRANSACTION	inappropriate_access_mode_for_branch_ transaction
25004	INAPPROPRIATE ISOLATION LEVEL FOR BRANCH TRANSACTION	inappropriate_isolation_level_for_branch_ transaction
25005	NO ACTIVE SQL TRANSACTION FOR BRANCH TRANSACTION	no_active_sql_transaction_for_branch_ transaction
25006	READ ONLY SQL TRANSACTION	read_only_sql_transaction
25007	SCHEMA AND DATA STATEMENT MIXING NOT SUPPORTED	schema_and_data_statement_mixing_not_ supported
25P01	NO ACTIVE SQL TRANSACTION	no_active_sql_transaction
25P02	IN FAILED SQL TRANSACTION	in_failed_sql_transaction
Class 26 — Invalid SQL Statement Name		
26000	INVALID SQL STATEMENT NAME	invalid_sql_statement_name
Class 27 — Triggered Data Change Violation		
27000	TRIGGERED DATA CHANGE VIOLATION	triggered_data_change_violation
Class 28 — Invalid Authorization Specification		
28000	INVALID AUTHORIZATION SPECIFICATION	invalid_authorization_specification
Class 2B — Dependent Privilege Descriptors Still Exist		

Error Code	Meaning	Constant
2B000	DEPENDENT PRIVILEGE DESCRIPTORS STILL EXIST	dependent_privilege_descriptors_still_exist
2BP01	DEPENDENT OBJECTS STILL EXIST	dependent_objects_still_exist
Class 2D — Invalid Transaction Termination		
2D000	INVALID TRANSACTION TERMINATION	invalid_transaction_termination
Class 2F — SQL Routine Exception		
2F000	SQL ROUTINE EXCEPTION	sql_routine_exception
2F005	FUNCTION EXECUTED NO RETURN STATEMENT	function_executed_no_return_statement
2F002	MODIFYING SQL DATA NOT PERMITTED	modifying_sql_data_not_permitted
2F003	PROHIBITED SQL STATEMENT ATTEMPTED	prohibited_sql_statement_attempted
2F004	READING SQL DATA NOT PERMITTED	reading_sql_data_not_permitted
Class 34 — Invalid Cursor Name		
34000	INVALID CURSOR NAME	invalid_cursor_name
Class 38 — External Routine Exception		
38000	EXTERNAL ROUTINE EXCEPTION	external_routine_exception
38001	CONTAINING SQL NOT PERMITTED	containing_sql_not_permitted
38002	MODIFYING SQL DATA NOT PERMITTED	modifying_sql_data_not_permitted
38003	PROHIBITED SQL STATEMENT ATTEMPTED	prohibited_sql_statement_attempted
38004	READING SQL DATA NOT PERMITTED	reading_sql_data_not_permitted
Class 39 — External Routine Invocation Exception		
39000	EXTERNAL ROUTINE INVOCATION EXCEPTION	external_routine_invocation_exception
39001	INVALID SQLSTATE RETURNED	invalid_sqlstate_returned
39004	NULL VALUE NOT ALLOWED	null_value_not_allowed
39P01	TRIGGER PROTOCOL VIOLATED	trigger_protocol_violated
39P02	SRF PROTOCOL VIOLATED	srf_protocol_violated
Class 3B — Savepoint Exception		
3B000	SAVEPOINT EXCEPTION	savepoint_exception
3B001	INVALID SAVEPOINT SPECIFICATION	invalid_savepoint_specification
Class 3D — Invalid Catalog Name		

Error Code	Meaning	Constant
3D000	INVALID CATALOG NAME	invalid_catalog_name
Class 3F — Invalid Schema Name		
3F000	INVALID SCHEMA NAME	invalid_schema_name
Class 40 — Transaction Rollback		
40000	TRANSACTION ROLLBACK	transaction_rollback
40002	TRANSACTION INTEGRITY CONSTRAINT VIOLATION	transaction_integrity_constraint_violation
40001	SERIALIZATION FAILURE	serialization_failure
40003	STATEMENT COMPLETION UNKNOWN	statement_completion_unknown
40P01	DEADLOCK DETECTED	deadlock_detected
Class 42 — Syntax Error or Access Rule Violation		
42000	SYNTAX ERROR OR ACCESS RULE VIOLATION	syntax_error_or_access_rule_violation
42601	SYNTAX ERROR	syntax_error
42501	INSUFFICIENT PRIVILEGE	insufficient_privilege
42846	CANNOT COERCE	cannot_coerce
42803	GROUPING ERROR	grouping_error
42830	INVALID FOREIGN KEY	invalid_foreign_key
42602	INVALID NAME	invalid_name
42622	NAME TOO LONG	name_too_long
42939	RESERVED NAME	reserved_name
42804	DATATYPE MISMATCH	datatype_mismatch
42P18	INDETERMINATE DATATYPE	indeterminate_datatype
42809	WRONG OBJECT TYPE	wrong_object_type
42703	UNDEFINED COLUMN	undefined_column
42883	UNDEFINED FUNCTION	undefined_function
42P01	UNDEFINED TABLE	undefined_table
42P02	UNDEFINED PARAMETER	undefined_parameter
42704	UNDEFINED OBJECT	undefined_object
42701	DUPLICATE COLUMN	duplicate_column
42P03	DUPLICATE CURSOR	duplicate_cursor
42P04	DUPLICATE DATABASE	duplicate_database
42723	DUPLICATE FUNCTION	duplicate_function
42P05	DUPLICATE PREPARED STATEMENT	duplicate_prepared_statement

Error Code	Meaning	Constant
42P06	DUPLICATE SCHEMA	duplicate_schema
42P07	DUPLICATE TABLE	duplicate_table
42712	DUPLICATE ALIAS	duplicate_alias
42710	DUPLICATE OBJECT	duplicate_object
42702	AMBIGUOUS COLUMN	ambiguous_column
42725	AMBIGUOUS FUNCTION	ambiguous_function
42P08	AMBIGUOUS PARAMETER	ambiguous_parameter
42P09	AMBIGUOUS ALIAS	ambiguous_alias
42P10	INVALID COLUMN REFERENCE	invalid_column_reference
42611	INVALID COLUMN DEFINITION	invalid_column_definition
42P11	INVALID CURSOR DEFINITION	invalid_cursor_definition
42P12	INVALID DATABASE DEFINITION	invalid_database_definition
42P13	INVALID FUNCTION DEFINITION	invalid_function_definition
42P14	INVALID PREPARED STATEMENT DEFINITION	invalid_prepared_statement_definition
42P15	INVALID SCHEMA DEFINITION	invalid_schema_definition
42P16	INVALID TABLE DEFINITION	invalid_table_definition
42P17	INVALID OBJECT DEFINITION	invalid_object_definition
Class 44 — WITH CHECK OPTION Violation		
44000	WITH CHECK OPTION VIOLATION	with_check_option_violation
Class 53 — Insufficient Resources		
53000	INSUFFICIENT RESOURCES	insufficient_resources
53100	DISK FULL	disk_full
53200	OUT OF MEMORY	out_of_memory
53300	TOO MANY CONNECTIONS	too_many_connections
Class 54 — Program Limit Exceeded		
54000	PROGRAM LIMIT EXCEEDED	program_limit_exceeded
54001	STATEMENT TOO COMPLEX	statement_too_complex
54011	TOO MANY COLUMNS	too_many_columns
54023	TOO MANY ARGUMENTS	too_many_arguments
Class 55 — Object Not In Prerequisite State		
55000	OBJECT NOT IN PREREQUISITE STATE	object_not_in_prerequisite_state
55006	OBJECT IN USE	object_in_use
55P02	CANT CHANGE RUNTIME PARAM	cant_change_runtime_param

Error Code	Meaning	Constant
55P03	LOCK NOT AVAILABLE	lock_not_available
Class 57 — Operator Intervention		
57000	OPERATOR INTERVENTION	operator_intervention
57014	QUERY CANCELED	query_canceled
57P01	ADMIN SHUTDOWN	admin_shutdown
57P02	CRASH SHUTDOWN	crash_shutdown
57P03	CANNOT CONNECT NOW	cannot_connect_now
Class 58 — System Error (errors external to Greenplum Database)		
58030	IO ERROR	io_error
58P01	UNDEFINED FILE	undefined_file
58P02	DUPLICATE FILE	duplicate_file
Class F0 — Configuration File Error		
F0000	CONFIG FILE ERROR	config_file_error
F0001	LOCK FILE EXISTS	lock_file_exists
Class P0 — PL/pgSQL Error		
P0000	PLPGSQL ERROR	plpgsql_error
P0001	RAISE EXCEPTION	raise_exception
P0002	NO DATA FOUND	no_data_found
P0003	TOO MANY ROWS	too_many_rows
Class XX — Internal Error		
XX000	INTERNAL ERROR	internal_error
XX001	DATA CORRUPTED	data_corrupted
XX002	INDEX CORRUPTED	index_corrupted

Routine System Maintenance Tasks

To keep a Greenplum Database system running efficiently, the database must be regularly cleared of expired data and the table statistics must be updated so that the query optimizer has accurate information.

Greenplum Database requires that certain tasks be performed regularly to achieve optimal performance. The tasks discussed here are required, but database administrators can automate them using standard UNIX tools such as `cron` scripts. An administrator sets up the appropriate scripts and checks that they execute successfully. See *Recommended Monitoring and Maintenance Tasks* for additional suggested maintenance activities you can implement to keep your Greenplum system running optimally.

Routine Vacuum and Analyze

The design of the MVCC transaction concurrency model used in Greenplum Database means that deleted or updated data rows still occupy physical space on disk even though they are not visible to new transactions. If your database has many updates and deletes, many expired rows exist and the space they use must be reclaimed with the `VACUUM` command. The `VACUUM` command also collects table-level

statistics, such as numbers of rows and pages, so it is also necessary to vacuum append-optimized tables, even when there is no space to reclaim from updated or deleted rows.

Vacuuming an append-optimized table follows a different process than vacuuming heap tables. On each segment, a new segment file is created and visible rows are copied into it from the current segment. When the segment file has been copied, the original is scheduled to be dropped and the new segment file is made available. This requires sufficient available disk space for a copy of the visible rows until the original segment file is dropped.

If the ratio of hidden rows to total rows in a segment file is less than a threshold value (10, by default), the segment file is not compacted. The threshold value can be configured with the `gp_appendonly_compaction_threshold` server configuration parameter. `VACUUM FULL` ignores the value of `gp_appendonly_compaction_threshold` and rewrites the segment file regardless of the ratio.

You can use the `__gp_aovisimap_compaction_info()` function in the *gp_toolkit* schema to investigate the effectiveness of a `VACUUM` operation on append-optimized tables.

For information about the `__gp_aovisimap_compaction_info()` function see, "Checking Append-Optimized Tables" in the *Greenplum Database Reference Guide*.

`VACUUM` can be disabled for append-optimized tables using the `gp_appendonly_compaction` server configuration parameter.

For details about vacuuming a database, see *Vacuuming the Database*.

For information about the `gp_appendonly_compaction_threshold` server configuration parameter and the `VACUUM` command, see the *Greenplum Database Reference Guide*.

Transaction ID Management

Greenplum's MVCC transaction semantics depend on comparing transaction ID (XID) numbers to determine visibility to other transactions. Transaction ID numbers are compared using modulo 2^{32} arithmetic, so a Greenplum system that runs more than about two billion transactions can experience transaction ID wraparound, where past transactions appear to be in the future. This means past transactions' outputs become invisible. Therefore, it is necessary to `VACUUM` every table in every database at least once per two billion transactions.

Greenplum Database assigns XID values only to transactions that involve DDL or DML operations, which are typically the only transactions that require an XID.

> **Important:** Greenplum Database monitors transaction IDs. If you do not vacuum the database regularly, Greenplum Database will generate a warning and error.

Greenplum Database issues the following warning when a significant portion of the transaction IDs are no longer available and before transaction ID wraparound occurs:

```
WARNING: database "database_name" must be vacuumed within
number_of_transactions transactions
```

When the warning is issued, a `VACUUM` operation is required. If a `VACUUM` operation is not performed, Greenplum Database stops creating transactions when it reaches a limit prior to when transaction ID wraparound occurs. Greenplum Database issues this error when it stops creating transactions to avoid possible data loss:

```
FATAL: database is not accepting commands to avoid
wraparound data loss in database "database_name"
```

The Greenplum Database configuration parameter `xid_warn_limit` controls when the warning is displayed. The parameter `xid_stop_limit` controls when Greenplum Database stops creating transactions.

Recovering from a Transaction ID Limit Error

When Greenplum Database reaches the `xid_stop_limit` transaction ID limit due to infrequent `VACUUM` maintenance, it becomes unresponsive. To recover from this situation, perform the following steps as database administrator:

1. Shut down Greenplum Database.
2. Temporarily lower the `xid_stop_limit` by 10,000,000.
3. Start Greenplum Database.
4. Run `VACUUM FREEZE` on all affected databases.
5. Reset the `xid_stop_limit` to its original value.
6. Restart Greenplum Database.

For information about the configuration parameters, see the *Greenplum Database Reference Guide*.

For information about transaction ID wraparound see the *PostgreSQL documentation*.

System Catalog Maintenance

Numerous database updates with `CREATE` and `DROP` commands increase the system catalog size and affect system performance. For example, running many `DROP TABLE` statements degrades the overall system performance due to excessive data scanning during metadata operations on catalog tables. The performance loss occurs between thousands to tens of thousands of `DROP TABLE` statements, depending on the system.

You should run a system catalog maintenance procedure regularly to reclaim the space occupied by deleted objects. If a regular procedure has not been run for a long time, you may need to run a more intensive procedure to clear the system catalog. This topic describes both procedures.

Regular System Catalog Maintenance

It is recommended that you periodically run `REINDEX` and `VACUUM` on the system catalog to clear the space that deleted objects occupy in the system indexes and tables. If regular database operations include numerous `DROP` statements, it is safe and appropriate to run a system catalog maintenance procedure with `VACUUM` daily at off-peak hours. You can do this while the system is available.

These are Greenplum Database system catalog maintenance steps.

1. Perform a `REINDEX` on the system catalog tables to rebuild the system catalog indexes. This removes bloat in the indexes and improves `VACUUM` performance.

   Note: When performing `REINDEX` on the system catalog tables, locking will occur on the tables and might have an impact on currently running queries. You can schedule the `REINDEX` operation during a period of low activity to avoid disrupting ongoing business operations.
2. Perform a `VACUUM` on the system catalog tables.
3. Perform an `ANALYZE` on the system catalog tables to update the catalog table statistics.

This example script performs a `REINDEX`, `VACUUM`, and `ANALYZE` of a Greenplum Database system catalog. In the script, replace *<database-name>* with a database name.

```
#!/bin/bash
DBNAME="<database-name>"
SYSTABLES="' pg_catalog.' || relname || ';' FROM pg_class a, pg_namespace b
WHERE a.relnamespace=b.oid AND b.nspname='pg_catalog' AND a.relkind='r'"

reindexdb --system -d $DBNAME
psql -tc "SELECT 'VACUUM' || $SYSTABLES" $DBNAME | psql -a $DBNAME
analyzedb -s pg_catalog -d $DBNAME
```

Note: If you are performing catalog maintenance during a maintenance period and you need to stop a process due to time constraints, run the Greenplum Database function `pg_cancel_backend(<PID>)` to safely stop the Greenplum Database process.

Intensive System Catalog Maintenance

If system catalog maintenance has not been performed in a long time, the catalog can become bloated with dead space; this causes excessively long wait times for simple metadata operations. A wait of more than two seconds to list user tables, such as with the `\d` metacommand from within `psql`, is an indication of catalog bloat.

If you see indications of system catalog bloat, you must perform an intensive system catalog maintenance procedure with `VACUUM FULL` during a scheduled downtime period. During this period, stop all catalog activity on the system; the `VACUUM FULL` system catalog maintenance procedure takes exclusive locks against the system catalog.

Running regular system catalog maintenance procedures can prevent the need for this more costly procedure.

These are steps for intensive system catalog maintenance.

1. Stop all catalog activity on the Greenplum Database system.
2. Perform a `REINDEX` on the system catalog tables to rebuild the system catalog indexes. This removes bloat in the indexes and improves `VACUUM` performance.
3. Perform a `VACUUM FULL` on the system catalog tables. See the following Note.
4. Perform an `ANALYZE` on the system catalog tables to update the catalog table statistics.

Note: The system catalog table `pg_attribute` is usually the largest catalog table. If the `pg_attribute` table is significantly bloated, a `VACUUM FULL` operation on the table might require a significant amount of time and might need to be performed separately. The presence of both of these conditions indicate a significantly bloated `pg_attribute` table that might require a long `VACUUM FULL` time:

- The `pg_attribute` table contains a large number of records.
- The diagnostic message for `pg_attribute` is `significant amount of bloat` in the `gp_toolkit.gp_bloat_diag` view.

Vacuum and Analyze for Query Optimization

Greenplum Database uses a cost-based query optimizer that relies on database statistics. Accurate statistics allow the query optimizer to better estimate selectivity and the number of rows that a query operation retrieves. These estimates help it choose the most efficient query plan. The `ANALYZE` command collects column-level statistics for the query optimizer.

You can run both `VACUUM` and `ANALYZE` operations in the same command. For example:

```
=# VACUUM ANALYZE mytable;
```

Running the `VACUUM ANALYZE` command might produce incorrect statistics when the command is run on a table with a significant amount of bloat (a significant amount of table disk space is occupied by deleted or obsolete rows). For large tables, the `ANALYZE` command calculates statistics from a random sample of rows. It estimates the number rows in the table by multiplying the average number of rows per page in the sample by the number of actual pages in the table. If the sample contains many empty pages, the estimated row count can be inaccurate.

For a table, you can view information about the amount of unused disk space (space that is occupied by deleted or obsolete rows) in the *gp_toolkit* view *gp_bloat_diag*. If the `bdidiag` column for a table contains the value `significant amount of bloat suspected`, a significant amount of table disk space consists of unused space. Entries are added to the *gp_bloat_diag* view after a table has been vacuumed.

To remove unused disk space from the table, you can run the command `VACUUM FULL` on the table. Due to table lock requirements, `VACUUM FULL` might not be possible until a maintenance period.

As a temporary workaround, run `ANALYZE` to compute column statistics and then run `VACUUM` on the table to generate an accurate row count. This example runs `ANALYZE` and then `VACUUM` on the *cust_info* table.

```
ANALYZE cust_info;
VACUUM cust_info;
```

Important: If you intend to execute queries on partitioned tables with GPORCA enabled (the default), you must collect statistics on the partitioned table root partition with the `ANALYZE` command. For information about GPORCA, see *Overview of GPORCA*.

Note: You can use the Greenplum Database utility `analyzedb` to update table statistics. Tables can be analyzed concurrently. For append optimized tables, `analyzedb` updates statistics only if the statistics are not current. See the *`analyzedb`* utility.

Routine Reindexing

For B-tree indexes, a freshly-constructed index is slightly faster to access than one that has been updated many times because logically adjacent pages are usually also physically adjacent in a newly built index. Reindexing older indexes periodically can improve access speed. If all but a few index keys on a page have been deleted, there will be wasted space on the index page. A reindex will reclaim that wasted space. In Greenplum Database it is often faster to drop an index (`DROP INDEX`) and then recreate it (`CREATE INDEX`) than it is to use the `REINDEX` command.

For table columns with indexes, some operations such as bulk updates or inserts to the table might perform more slowly because of the updates to the indexes. To enhance performance of bulk operations on tables with indexes, you can drop the indexes, perform the bulk operation, and then re-create the index.

Managing Greenplum Database Log Files

- *Database Server Log Files*
- *Management Utility Log Files*

Database Server Log Files

Greenplum Database log output tends to be voluminous, especially at higher debug levels, and you do not need to save it indefinitely. Administrators should purge older log files periodically.

Greenplum Database by default has log file rotation enabled for the master and segment database logs. Log files are created in the `pg_log` subdirectory of the master and each segment data directory using the following naming convention: `gpdb-`*`YYYY-MM-DD_hhmmss`*`.csv`. Administrators need to implement scripts or programs to periodically clean up old log files in the `pg_log` directory of the master and each segment instance.

Log rotation can be triggered by the size of the current log file or the age of the current log file. The `log_rotation_size` configuration parameter sets the size of an individual log file that triggers log rotation. When the log file size is equal to or greater than the specified size, the file is closed and a new log file is created. The `log_rotation_size` value is specified in kilobytes. The default is 1048576 kilobytes, or 1GB. If `log_rotation_size` is set to 0, size-based rotation is disabled.

The `log_rotation_age` configuration parameter specifies the age of a log file that triggers rotation. When the specified amount of time has elapsed since the log file was created, the file is closed and a new log file is created. The default `log_rotation_age`, 1d, creates a new log file 24 hours after the current log file was created. If `log_rotation_age` is set to 0, time-based rotation is disabled.

For information about viewing the database server log files, see *Viewing the Database Server Log Files*.

Management Utility Log Files

Log files for the Greenplum Database management utilities are written to `~/gpAdminLogs` by default. The naming convention for management log files is:

```
script_name_date.log
```

The log entry format is:

```
timestamp:utility:host:user:[INFO|WARN|FATAL]:message
```

The log file for a particular utility execution is appended to its daily log file each time that utility is run.

Recommended Monitoring and Maintenance Tasks

This section lists monitoring and maintenance activities recommended to ensure high availability and consistent performance of your Greenplum Database cluster.

The tables in the following sections suggest activities that a Greenplum System Administrator can perform periodically to ensure that all components of the system are operating optimally. Monitoring activities help you to detect and diagnose problems early. Maintenance activities help you to keep the system up-to-date and avoid deteriorating performance, for example, from bloated system tables or diminishing free disk space.

It is not necessary to implement all of these suggestions in every cluster; use the frequency and severity recommendations as a guide to implement measures according to your service requirements.

Database State Monitoring Activities

Table 29: Database State Monitoring Activities

Activity	Procedure	Corrective Actions
List segments that are currently down. If any rows are returned, this should generate a warning or alert. Recommended frequency: run every 5 to 10 minutes Severity: IMPORTANT	Run the following query in the `postgres` database: `SELECT * FROM gp_segment_configuration WHERE status <> 'u';`	If the query returns any rows, follow these steps to correct the problem: 1. Verify that the hosts with down segments are responsive. 2. If hosts are OK, check the `pg_log` files for the primaries and mirrors of the down segments to discover the root cause of the segments going down. 3. If no unexpected errors are found, run the `gprecoverseg` utility to bring the segments back online.

Activity	Procedure	Corrective Actions
Check for segments that are currently in change tracking mode. If any rows are returned, this should generate a warning or alert. Recommended frequency: run every 5 to 10 minutes Severity: IMPORTANT	Execute the following query in the `postgres` database: `SELECT * FROM gp_segment_configuration WHERE mode = 'c';`	If the query returns any rows, follow these steps to correct the problem: 1. Verify that hosts with down segments are responsive. 2. If hosts are OK, check the `pg_log` files for the primaries and mirrors of the down segments to determine the root cause of the segments going down. 3. If no unexpected errors are found, run the `gprecoverseg` utility to bring the segments back online.
Check for segments that are currently re-syncing. If rows are returned, this should generate a warning or alert. Recommended frequency: run every 5 to 10 minutes Severity: IMPORTANT	Execute the following query in the `postgres` database: `SELECT * FROM gp_segment_configuration WHERE mode = 'r';`	When this query returns rows, it implies that the segments are in the process of being re-synched. If the state does not change from 'r' to 's', then check the `pg_log` files from the primaries and mirrors of the affected segments for errors.
Check for segments that are not operating in their optimal role. If any segments are found, the cluster may not be balanced. If any rows are returned this should generate a warning or alert. Recommended frequency: run every 5 to 10 minutes Severity: IMPORTANT	Execute the following query in the `postgres` database: `SELECT * FROM gp_segment_configuration WHERE preferred_role <> role;`	When the segments are not running in their preferred role, hosts have uneven numbers of primary segments on each host, implying that processing is skewed. Wait for a potential window and restart the database to bring the segments into their preferred roles.
Run a distributed query to test that it runs on all segments. One row should be returned for each primary segment. Recommended frequency: run every 5 to 10 minutes Severity: CRITICAL	Execute the following query in the `postgres` database: `SELECT gp_segment_id, count(*) FROM gp_dist_random('pg_class') GROUP BY 1;`	If this query fails, there is an issue dispatching to some segments in the cluster. This is a rare event. Check the hosts that are not able to be dispatched to ensure there is no hardware or networking issue.
Test the state of master mirroring on a Greenplum Database 4.2 or earlier cluster. If the value is "Not Synchronized", raise an alert or warning. Recommended frequency: run every 5 to 10 minutes Severity: IMPORTANT	Execute the following query in the `postgres` database: `SELECT summary_state FROM gp_master_mirroring;`	Check the `pg_log` from the master and standby master for errors. If there are no unexpected errors and the machines are up, run the `gpinitstandby` utility to bring the standby online. This requires a database restart on GPDB 4.2 and earlier.

Activity	Procedure	Corrective Actions
Test the state of master mirroring on Greenplum Database. If the value is not "STREAMING", raise an alert or warning. Recommended frequency: run every 5 to 10 minutes Severity: IMPORTANT	Run the following `psql` command: `psql *dbname* -c 'SELECT pid, state FROM pg_stat_replication;'`	Check the `pg_log` file from the master and standby master for errors. If there are no unexpected errors and the machines are up, run the `gpinitstandby` utility to bring the standby online.
Perform a basic check to see if the master is up and functioning. Recommended frequency: run every 5 to 10 minutes Severity: CRITICAL	Run the following query in the `postgres` database: `SELECT count(*) FROM gp_segment_configuration;`	If this query fails the active master may be down. Try again several times and then inspect the active master manually. If the active master is down, reboot or power cycle the active master to ensure no processes remain on the active master and then trigger the activation of the standby master.

Database Alert Log Monitoring

Table 30: Database Alert Log Monitoring Activities

Activity	Procedure	Corrective Actions
Check for FATAL and ERROR log messages from the system. Recommended frequency: run every 15 minutes Severity: WARNING *This activity and the next are two methods for monitoring messages in the log_alert_history table. It is only necessary to set up one or the other.*	Run the following query in the `gpperfmon` database: `SELECT * FROM log_alert_history WHERE logseverity in ('FATAL', 'ERROR') AND logtime > (now() - interval '15 minutes');`	Send an alert to the DBA to analyze the alert. You may want to add additional filters to the query to ignore certain messages of low interest.

Hardware and Operating System Monitoring

Table 31: Hardware and Operating System Monitoring Activities

Activity	Procedure	Corrective Actions
Check disk space usage on volumes used for Greenplum Database data storage and the OS. Recommended frequency: every 5 to 30 minutes Severity: CRITICAL	Set up a disk space check. • Set a threshold to raise an alert when a disk reaches a percentage of capacity. The recommended threshold is 75% full. • It is not recommended to run the system with capacities approaching 100%.	Free space on the system by removing some data or files.
Check for errors or dropped packets on the network interfaces. Recommended frequency: hourly Severity: IMPORTANT	Set up a network interface checks.	Work with network and OS teams to resolve errors.
Check for RAID errors or degraded RAID performance. Recommended frequency: every 5 minutes Severity: CRITICAL	Set up a RAID check.	• Replace failed disks as soon as possible. • Work with system administration team to resolve other RAID or controller errors as soon as possible.
Check for adequate I/O bandwidth and I/O skew. Recommended frequency: when create a cluster or when hardware issues are suspected.	Run the Greenplum `gpcheckperf` utility.	The cluster may be under-specified if data transfer rates are not similar to the following: • 2GB per second disk read • 1 GB per second disk write • 10 Gigabit per second network read and write If transfer rates are lower than expected, consult with your data architect regarding performance expectations. If the machines on the cluster display an uneven performance profile, work with the system administration team to fix faulty machines.

Catalog Monitoring

Table 32: Catalog Monitoring Activities

Activity	Procedure	Corrective Actions
Run catalog consistency checks to ensure the catalog on each host in the cluster is consistent and in a good state. Recommended frequency: weekly Severity: IMPORTANT	Run the Greenplum `gpcheckcat` utility in each database: `gpcheckcat -O`	Run repair scripts for any issues detected.
Check for `pg_class` entries that have no corresponding `pg_attribute` entry. Recommended frequency: monthly Severity: IMPORTANT	During a downtime, with no users on the system, run the Greenplum `gpcheckcat` utility in each database: `gpcheckcat -R pgclass`	Run the repair scripts for any issues identified.
Check for leaked temporary schema and missing schema definition. Recommended frequency: monthly Severity: IMPORTANT	During a downtime, with no users on the system, run the Greenplum `gpcheckcat` utility in each database: `gpcheckcat -R namespace`	Run the repair scripts for any issues identified.
Check constraints on randomly distributed tables. Recommended frequency: monthly Severity: IMPORTANT	During a downtime, with no users on the system, run the Greenplum `gpcheckcat` utility in each database: `gpcheckcat -R distribution_policy`	Run the repair scripts for any issues identified.
Check for dependencies on non-existent objects. Recommended frequency: monthly Severity: IMPORTANT	During a downtime, with no users on the system, run the Greenplum `gpcheckcat` utility in each database: `gpcheckcat -R dependency`	Run the repair scripts for any issues identified.

Data Maintenance

Table 33: Data Maintenance Activities

Activity	Procedure	Corrective Actions
Check for missing statistics on tables.	Check the `gp_stats_missing` view in each database: `SELECT * FROM gp_toolkit.gp_stats_missing;`	Run `ANALYZE` on tables that are missing statistics.
Check for tables that have bloat (dead space) in data files that cannot be recovered by a regular `VACUUM` command. Recommended frequency: weekly or monthly Severity: WARNING	Check the `gp_bloat_diag` view in each database: `SELECT * FROM gp_toolkit.gp_bloat_diag;`	`VACUUM FULL` acquires an `ACCESS EXCLUSIVE` lock on tables. Run `VACUUM FULL` during a time when users and applications do not require access to the tables, such as during a time of low activity, or during a maintenance window.

Database Maintenance

Table 34: Database Maintenance Activities

Activity	Procedure	Corrective Actions
Mark deleted rows in heap tables so that the space they occupy can be reused. Recommended frequency: daily Severity: CRITICAL	Vacuum user tables: `VACUUM <table>;`	Vacuum updated tables regularly to prevent bloating.
Update table statistics. Recommended frequency: after loading data and before executing queries Severity: CRITICAL	Analyze user tables. You can use the `analyzedb` management utility: `analyzedb -d <database> -a`	Analyze updated tables regularly so that the optimizer can produce efficient query execution plans.
Backup the database data. Recommended frequency: daily, or as required by your backup plan Severity: CRITICAL	Run the `gpbackup` utility to create a backup of the master and segment databases in parallel.	Best practice is to have a current backup ready in case the database must be restored.

Activity	Procedure	Corrective Actions

Vacuum, reindex, and analyze system catalogs to maintain an efficient catalog.
Recommended frequency: weekly, or more often if database objects are created and dropped frequently
	1. VACUUM the system tables in each database.
	2. Run REINDEX SYSTEM in each database, or use the reindexdb command-line utility with the -s option:
	`reindexdb -s <database>`
	3. ANALYZE each of the system tables:
	`analyzedb -s pg_catalog -d <database>`
		The optimizer retrieves information from the system tables to create query plans. If system tables and indexes are allowed to become bloated over time, scanning the system tables increases query execution time. It is important to run ANALYZE after reindexing, because REINDEX leaves indexes with no statistics.

Patching and Upgrading

Table 35: Patch and Upgrade Activities

Activity	Procedure	Corrective Actions
Ensure any bug fixes or enhancements are applied to the kernel. Recommended frequency: at least every 6 months Severity: IMPORTANT	Follow the vendor's instructions to update the Linux kernel.	Keep the kernel current to include bug fixes and security fixes, and to avoid difficult future upgrades.
Install Greenplum Database minor releases, for example 5.0.*x*. Recommended frequency: quarterly Severity: IMPORTANT	Follow upgrade instructions in the Greenplum Database *Release Notes*. Always upgrade to the latest in the series.	Keep the Greenplum Database software current to incorporate bug fixes, performance enhancements, and feature enhancements into your Greenplum cluster.

Managing Greenplum Database Access

Securing Greenplum Database includes protecting access to the database through network configuration, database user authentication, and encryption.

This section contains the following topics:

Configuring Client Authentication

This topic explains how to configure client connections and authentication for Greenplum Database.

When a Greenplum Database system is first initialized, the system contains one predefined *superuser* role. This role will have the same name as the operating system user who initialized the Greenplum Database system. This role is referred to as `gpadmin`. By default, the system is configured to only allow local connections to the database from the `gpadmin` role. If you want to allow any other roles to connect, or if you want to allow connections from remote hosts, you have to configure Greenplum Database to allow such connections. This section explains how to configure client connections and authentication to Greenplum Database.

Allowing Connections to Greenplum Database

Client access and authentication is controlled by the standard PostgreSQL host-based authentication file, `pg_hba.conf`. For detailed information about this file, see *The pg_hba.conf File* in the PostgreSQL documentation.

In Greenplum Database, the `pg_hba.conf` file of the master instance controls client access and authentication to your Greenplum Database system. The Greenplum Database segments also have `pg_hba.conf` files, but these are already correctly configured to allow only client connections from the master host. The segments never accept outside client connections, so there is no need to alter the `pg_hba.conf` file on segments.

The general format of the `pg_hba.conf` file is a set of records, one per line. Greenplum Database ignores blank lines and any text after the # comment character. A record consists of a number of fields that are separated by spaces or tabs. Fields can contain white space if the field value is quoted. Records cannot be continued across lines. Each remote client access record has the following format:

```
host   database   role   address   authentication-method
```

Each UNIX-domain socket access record is in this format:

```
local   database   role   authentication-method
```

The following table describes meaning of each field.

Table 36: pg_hba.conf Fields

Field	Description
local	Matches connection attempts using UNIX-domain sockets. Without a record of this type, UNIX-domain socket connections are disallowed.

Field	Description
host	Matches connection attempts made using TCP/IP. Remote TCP/IP connections will not be possible unless the server is started with an appropriate value for the `listen_addresses` server configuration parameter.
hostssl	Matches connection attempts made using TCP/IP, but only when the connection is made with SSL encryption. SSL must be enabled at server start time by setting the `ssl` server configuration parameter.
hostnossl	Matches connection attempts made over TCP/IP that do not use SSL.
database	Specifies which database names this record matches. The value `all` specifies that it matches all databases. Multiple database names can be supplied by separating them with commas. A separate file containing database names can be specified by preceding the file name with a @.
role	Specifies which database role names this record matches. The value `all` specifies that it matches all roles. If the specified role is a group and you want all members of that group to be included, precede the role name with a +. Multiple role names can be supplied by separating them with commas. A separate file containing role names can be specified by preceding the file name with a @.

Field	Description
address	Specifies the client machine addresses that this record matches. This field can contain an IP address, an IP address range, or a host name. An IP address range is specified using standard numeric notation for the range's starting address, then a slash (/) and a CIDR mask length. The mask length indicates the number of high-order bits of the client IP address that must match. Bits to the right of this should be zero in the given IP address. There must not be any white space between the IP address, the /, and the CIDR mask length. Typical examples of an IPv4 address range specified this way are `172.20.143.89/32` for a single host, or `172.20.143.0/24` for a small network, or `10.6.0.0/16` for a larger one. An IPv6 address range might look like `::1/128` for a single host (in this case the IPv6 loopback address) or `fe80::7a31:c1ff:0000:0000/96` for a small network. `0.0.0.0/0` represents all IPv4 addresses, and `::0/0` represents all IPv6 addresses. To specify a single host, use a mask length of 32 for IPv4 or 128 for IPv6. In a network address, do not omit trailing zeroes. An entry given in IPv4 format will match only IPv4 connections, and an entry given in IPv6 format will match only IPv6 connections, even if the represented address is in the IPv4-in-IPv6 range. **Note:** Entries in IPv6 format will be rejected if the host system C library does not have support for IPv6 addresses. If a host name is specified (an address that is not an IP address or IP range is treated as a host name), that name is compared with the result of a reverse name resolution of the client IP address (for example, reverse DNS lookup, if DNS is used). Host name comparisons are case insensitive. If there is a match, then a forward name resolution (for example, forward DNS lookup) is performed on the host name to check whether any of the addresses it resolves to are equal to the client IP address. If both directions match, then the entry is considered to match. Some host name databases allow associating an IP address with multiple host names, but the operating system only returns one host name when asked to resolve an IP address. The host name that is used in `pg_hba.conf` must be the one that the address-to-name resolution of the client IP address returns, otherwise the line will not be considered a match. When host names are specified in `pg_hba.conf`, you should ensure that name resolution is reasonably fast. It can be of advantage to set up a local name resolution cache such as `nscd`. Also, you can enable the server configuration parameter `log_hostname` to see the client host name instead of the IP address in the log.
IP-address IP-mask	These fields can be used as an alternative to the CIDR address notation. Instead of specifying the mask length, the actual mask is specified in a separate column. For example, `255.0.0.0` represents an IPv4 CIDR mask length of 8, and `255.255.255.255` represents a CIDR mask length of 32.
authentication-method	Specifies the authentication method to use when connecting. Greenplum supports the *authentication methods* supported by PostgreSQL 9.4.

Caution: For a more secure system, consider removing records for remote connections that use trust authentication from the `pg_hba.conf` file. Trust authentication grants any user who can connect to the server access to the database using any role they specify. You can safely replace

trust authentication with ident authentication for local UNIX-socket connections. You can also use ident authentication for local and remote TCP clients, but the client host must be running an ident service and you must trust the integrity of that machine.

Editing the pg_hba.conf File

Initially, the `pg_hba.conf` file is set up with generous permissions for the gpadmin user and no database access for other Greenplum Database roles. You will need to edit the `pg_hba.conf` file to enable users' access to databases and to secure the gpadmin user. Consider removing entries that have trust authentication, since they allow anyone with access to the server to connect with any role they choose. For local (UNIX socket) connections, use ident authentication, which requires the operating system user to match the role specified. For local and remote TCP connections, ident authentication requires the client's host to run an indent service. You can install an ident service on the master host and then use ident authentication for local TCP connections, for example `127.0.0.1/28`. Using ident authentication for remote TCP connections is less secure because it requires you to trust the integrity of the ident service on the client's host.

This example shows how to edit the `pg_hba.conf` file of the master to allow remote client access to all databases from all roles using encrypted password authentication.

Editing pg_hba.conf

1. Open the file `$MASTER_DATA_DIRECTORY/pg_hba.conf` in a text editor.
2. Add a line to the file for each type of connection you want to allow. Records are read sequentially, so the order of the records is significant. Typically, earlier records will have tight connection match parameters and weaker authentication methods, while later records will have looser match parameters and stronger authentication methods. For example:

```
# allow the gpadmin user local access to all databases
# using ident authentication
local   all   gpadmin   ident         sameuser
host    all   gpadmin   127.0.0.1/32  ident
host    all   gpadmin   ::1/128       ident
# allow the 'dba' role access to any database from any
# host with IP address 192.168.x.x and use md5 encrypted
# passwords to authenticate the user
# Note that to use SHA-256 encryption, replace md5 with
# password in the line below
host    all   dba   192.168.0.0/32  md5
# allow all roles access to any database from any
# host and use ldap to authenticate the user. Greenplum role
# names must match the LDAP common name.
host    all   all   192.168.0.0/32  ldap ldapserver=usldap1 ldapport=1389
 ldapprefix="cn=" ldapsuffix=",ou=People,dc=company,dc=com"
```

3. Save and close the file.
4. Reload the `pg_hba.conf` configuration file for your changes to take effect:

```
$ gpstop -u
```

Note: Note that you can also control database access by setting object privileges as described in *Managing Object Privileges*. The `pg_hba.conf` file just controls who can initiate a database session and how those connections are authenticated.

Limiting Concurrent Connections

Greenplum Database allocates some resources on a per-connection basis, so setting the maximum number of connections allowed is recommended.

To limit the number of active concurrent sessions to your Greenplum Database system, you can configure the `max_connections` server configuration parameter. This is a *local* parameter, meaning that you must set it in the `postgresql.conf` file of the master, the standby master, and each segment instance (primary and mirror). The recommended value of `max_connections` on segments is 5-10 times the value on the master.

When you set `max_connections`, you must also set the dependent parameter `max_prepared_transactions`. This value must be at least as large as the value of `max_connections` on the master, and segment instances should be set to the same value as the master.

For example:

- In `$MASTER_DATA_DIRECTORY/postgresql.conf` (including standby master):

```
max_connections=100
max_prepared_transactions=100
```

- In *`SEGMENT_DATA_DIRECTORY`*`/postgresql.conf` for all segment instances:

```
max_connections=500
max_prepared_transactions=100
```

The following steps set the parameter values with the Greenplum Database utility `gpconfig`.

For information about `gpconfig`, see the *Greenplum Database Utility Guide*.

To change the number of allowed connections

1. Log into the Greenplum Database master host as the Greenplum Database administrator and source the file `$GPHOME/greenplum_path.sh`.
2. Set the value of the `max_connections` parameter. This `gpconfig` command sets the value on the segments to 1000 and the value on the master to 200.

   ```
   $ gpconfig -c max_connections -v 1000 -m 200
   ```

   The value on the segments must be greater than the value on the master. The recommended value of `max_connections` on segments is 5-10 times the value on the master.
3. Set the value of the `max_prepared_transactions` parameter. This `gpconfig` command sets the value to 200 on the master and all segments.

   ```
   $ gpconfig -c max_prepared_transactions -v 200
   ```

   The value of `max_prepared_transactions` must be greater than or equal to `max_connections` on the master.
4. Stop and restart your Greenplum Database system.

   ```
   $ gpstop -r
   ```

5. You can check the value of parameters on the master and segments with the `gpconfig -s` option. This `gpconfig` command displays the values of the `max_connections` parameter.

   ```
   $ gpconfig -s max_connections
   ```

Note: Raising the values of these parameters may cause Greenplum Database to request more shared memory. To mitigate this effect, consider decreasing other memory-related parameters such as `gp_cached_segworkers_threshold`.

Encrypting Client/Server Connections

Enable SSL for client connections to Greenplum Database to encrypt the data passed over the network between the client and the database.

Greenplum Database has native support for SSL connections between the client and the master server. SSL connections prevent third parties from snooping on the packets, and also prevent man-in-the-middle attacks. SSL should be used whenever the client connection goes through an insecure link, and must be used whenever client certificate authentication is used.

Enabling Greenplum Database in SSL mode requires the following items.

- OpenSSL installed on both the client and the master server hosts (master and standby master).
- The SSL files `server.key` (server private key) and `server.crt` (server certificate) should be correctly generated for the master host and standby master host.
    - The private key should not be protected with a passphrase. The server does not prompt for a passphrase for the private key, and Greenplum Database start up fails with an error if one is required.
    - On a production system, there should be a key and certificate pair for the master host and a pair for the standby master host with a subject CN (Common Name) for the master host and standby master host.

    A self-signed certificate can be used for testing, but a certificate signed by a certificate authority (CA) should be used in production, so the client can verify the identity of the server. Either a global or local CA can be used. If all the clients are local to the organization, a local CA is recommended.
- Ensure that Greenplum Database can access `server.key` and `server.crt`, and any additional authentication files such as `root.crt` (for trusted certificate authorities). When starting in SSL mode, the Greenplum Database master looks for `server.key` and `server.crt`. As the default, Greenplum Database does not start if the files are not in the master data directory (`$MASTER_DATA_DIRECTORY`). Also, if you use other SSL authentication files such as `root.crt` (trusted certificate authorities), the files must be on the master host.

    If Greenplum Database master mirroring is enabled with SSL client authentication, SSL authentication files must be on both the master host and standby master host and *should not be placed* in the default directory `$MASTER_DATA_DIRECTORY`. When master mirroring is enabled, an `initstandby` operation copies the contents of the `$MASTER_DATA_DIRECTORY` from the master to the standby master and the incorrect SSL key, and cert files (the master files, and not the standby master files) will prevent standby master start up.

    You can specify a different directory for the location of the SSL server files with the `postgresql.conf` parameters `sslcert`, `sslkey`, `sslrootcert`, and `sslcrl`. For more information about the parameters, see *SSL Client Authentication* in the *Security Configuration Guide*.

Greenplum Database can be started with SSL enabled by setting the server configuration parameter `ssl=on` in the `postgresql.conf` file on the master and standby master hosts. This `gpconfig` command sets the parameter:

```
gpconfig -c ssl -m on -v off
```

Setting the parameter requires a server restart. This command restarts the system: `gpstop -ra`.

Creating a Self-signed Certificate without a Passphrase for Testing Only

To create a quick self-signed certificate for the server for testing, use the following OpenSSL command:

```
# openssl req -new -text -out server.req
```

Enter the information requested by the prompts. Be sure to enter the local host name as *Common Name*. The challenge password can be left blank.

The program will generate a key that is passphrase protected, and does not accept a passphrase that is less than four characters long.

To use this certificate with Greenplum Database, remove the passphrase with the following commands:

```
# openssl rsa -in privkey.pem -out server.key
# rm privkey.pem
```

Enter the old passphrase when prompted to unlock the existing key.

Then, enter the following command to turn the certificate into a self-signed certificate and to copy the key and certificate to a location where the server will look for them.

```
# openssl req -x509 -in server.req -text -key server.key -out server.crt
```

Finally, change the permissions on the key with the following command. The server will reject the file if the permissions are less restrictive than these.

```
# chmod og-rwx server.key
```

For more details on how to create your server private key and certificate, refer to the *OpenSSL documentation*.

Using LDAP Authentication with TLS/SSL

You can control access to Greenplum Database with an LDAP server and, optionally, secure the connection with encryption by adding parameters to `pg_hba.conf` file entries.

Greenplum Database supports LDAP authentication with the TLS/SSL protocol to encrypt communication with an LDAP server:

- LDAP authentication with STARTTLS and TLS protocol – STARTTLS starts with a clear text connection (no encryption) and upgrades it to a secure connection (with encryption).
- LDAP authentication with a secure connection and TLS/SSL (LDAPS) – Greenplum Database uses the TLS or SSL protocol based on the protocol that is used by the LDAP server.

If no protocol is specified, Greenplum Database communicates with the LDAP server with a clear text connection.

To use LDAP authentication, the Greenplum Database master host must be configured as an LDAP client. See your LDAP documentation for information about configuring LDAP clients.

Enabing LDAP Authentication with STARTTLS and TLS

To enable STARTTLS with the TLS protocol, in the pg_hba.conf file, add an `ldap` line and specify the `ldaptls` parameter with the value 1. The default port is 389. In this example, the authentication method parameters include the `ldaptls` parameter.

```
ldap ldapserver=myldap.com ldaptls=1 ldapprefix="uid="
 ldapsuffix=",ou=People,dc=example,dc=com"
```

Specify a non-default port with the `ldapport` parameter. In this example, the authentication method includes the `ldaptls` parameter and the `ldapport` parameter to specify the port 550.

```
ldap ldapserver=myldap.com ldaptls=1 ldapport=500 ldapprefix="uid="
 ldapsuffix=",ou=People,dc=example,dc=com"
```

Enabing LDAP Authentication with a Secure Connection and TLS/SSL

To enable a secure connection with TLS/SSL, add `ldaps://` as the prefix to the LDAP server name specified in the `ldapserver` parameter. The default port is 636.

This example `ldapserver` parameter specifies a secure connection and the TLS/SSL protocol for the LDAP server `myldap.com`.

```
ldapserver=ldaps://myldap.com
```

To specify a non-default port, add a colon (:) and the port number after the LDAP server name. This example `ldapserver` parameter includes the `ldaps://` prefix and the non-default port 550.

```
ldapserver=ldaps://myldap.com:550
```

Configuring Authentication with a System-wide OpenLDAP System

If you have a system-wide OpenLDAP system and logins are configured to use LDAP with TLS or SSL in the pg_hba.conf file, logins may fail with the following message:

```
could not start LDAP TLS session: error code '-11'
```

To use an existing OpenLDAP system for authentication, Greenplum Database must be set up to use the LDAP server's CA certificate to validate user certificates. Follow these steps on both the master and standby hosts to configure Greenplum Database:

1. Copy the base64-encoded root CA chain file from the Active Directory or LDAP server to the Greenplum Database master and standby master hosts. This example uses the directory `/etc/pki/tls/certs`.
2. Change to the directory where you copied the CA certificate file and, as the root user, generate the hash for OpenLDAP:

   ```
   # cd /etc/pki/tls/certs
   # openssl x509 -noout -hash -in <ca-certificate-file>
   # ln -s <ca-certificate-file> <ca-certificate-file>.0
   ```

3. Configure an OpenLDAP configuration file for Greenplum Database with the CA certificate directory and certificate file specified.

   As the root user, edit the OpenLDAP configuration file `/etc/openldap/ldap.conf`:

   ```
   SASL_NOCANON on
    URI ldaps://ldapA.example.priv ldaps://ldapB.example.priv ldaps://
   ldapC.example.priv
    BASE dc=example,dc=priv
    TLS_CACERTDIR /etc/pki/tls/certs
    TLS_CACERT /etc/pki/tls/certs/<ca-certificate-file>
   ```

   Note: For certificate validation to succeed, the hostname in the certificate must match a hostname in the URI property. Otherwise, you must also add `TLS_REQCERT allow` to the file.

4. As the gpadmin user, edit `/usr/local/greenplum-db/greenplum_path.sh` and add the following line.

```
export LDAPCONF=/etc/openldap/ldap.conf
```

Notes

Greenplum Database logs an error if the following are specified in an `pg_hba.conf` file entry:

- If both the `ldaps://` prefix and the `ldaptls=1` parameter are specified.
- If both the `ldaps://` prefix and the `ldapport` parameter are specified.

Enabling encrypted communication for LDAP authentication only encrypts the communication between Greenplum Database and the LDAP server.

See *Encrypting Client/Server Connections* for information about encrypting client connections.

Examples

These are example entries from an `pg_hba.conf` file.

This example specifies LDAP authentication with no encryption between Greenplum Database and the LDAP server.

```
host all plainuser 0.0.0.0/0 ldap ldapserver=myldap.com ldapprefix="uid="
 ldapsuffix=",ou=People,dc=example,dc=com"
```

This example specifies LDAP authentication with the STARTTLS and TLS protocol between Greenplum Database and the LDAP server.

```
host all tlsuser 0.0.0.0/0 ldap ldapserver=myldap.com ldaptls=1
 ldapprefix="uid=" ldapsuffix=",ou=People,dc=example,dc=com"
```

This example specifies LDAP authentication with a secure connection and TLS/SSL protocol between Greenplum Database and the LDAP server.

```
host all ldapsuser 0.0.0.0/0 ldap ldapserver=ldaps://myldap.com
 ldapprefix="uid=" ldapsuffix=",ou=People,dc=example,dc=com"
```

Using Kerberos Authentication

You can control access to Greenplum Database with a Kerberos authentication server.

Greenplum Database supports the Generic Security Service Application Program Interface (GSSAPI) with Kerberos authentication. GSSAPI provides automatic authentication (single sign-on) for systems that support it. You specify the Greenplum Database users (roles) that require Kerberos authentication in the Greenplum Database configuration file `pg_hba.conf`. The login fails if Kerberos authentication is not available when a role attempts to log in to Greenplum Database.

Kerberos provides a secure, encrypted authentication service. It does not encrypt data exchanged between the client and database and provides no authorization services. To encrypt data exchanged over the network, you must use an SSL connection. To manage authorization for access to Greenplum databases and objects such as schemas and tables, you use settings in the `pg_hba.conf` file and privileges given to Greenplum Database users and roles within the database. For information about managing authorization privileges, see *Managing Roles and Privileges*.

For more information about Kerberos, see *http://web.mit.edu/kerberos/*.

Prerequisites

Before configuring Kerberos authentication for Greenplum Database, ensure that:

- You can identify the KDC server you use for Kerberos authentication and the Kerberos realm for your Greenplum Database system. If you have not yet configured your MIT Kerberos KDC server, see *Installing and Configuring a Kerberos KDC Server* for example instructions.
- System time on the Kerberos Key Distribution Center (KDC) server and Greenplum Database master is synchronized. (For example, install the `ntp` package on both servers.)
- Network connectivity exists between the KDC server and the Greenplum Database master host.
- Java 1.7.0_17 or later is installed on all Greenplum Database hosts. Java 1.7.0_17 is required to use Kerberos-authenticated JDBC on Red Hat Enterprise Linux 6.x or 7.x.

Procedure

Following are the tasks to complete to set up Kerberos authentication for Greenplum Database.

- *Creating Greenplum Database Principals in the KDC Database*
- *Installing the Kerberos Client on the Master Host*
- *Configuring Greenplum Database to use Kerberos Authentication*
- *Mapping Kerberos Principals to Greenplum Database Roles*
- *Configuring JDBC Kerberos Authentication for Greenplum Database*
- *Configuring Kerberos for Linux Clients*
- *Configuring Kerberos For Windows Clients*

Creating Greenplum Database Principals in the KDC Database

Create a service principal for the Greenplum Database service and a Kerberos admin principal that allows managing the KDC database as the gpadmin user.

1. Log in to the Kerberos KDC server as the root user.

   ```
   $ ssh root@<kdc-server>
   ```

2. Create a principal for the Greenplum Database service.

   ```
   # kadmin.local -q "addprinc -randkey postgres/mdw@GPDB.KRB"
   ```

   The `-randkey` option prevents the command from prompting for a password.

   The `postgres` part of the principal names matches the value of the Greenplum Database `krb_srvname` server configuration parameter, which is `postgres` by default.

   The host name part of the principal name must match the output of the `hostname` command on the Greenplum Database master host. If the `hostname` command shows the fully qualified domain name (FQDN), use it in the principal name, for example `postgres/mdw.example.com@GPDB.KRB`.

   The `GPDB.KRB` part of the principal name is the Kerberos realm name.

3. Create a principal for the gpadmin/admin role.

   ```
   # kadmin.local -q "addprinc gpadmin/admin@GPDB.KRB"
   ```

   This principal allows you to manage the KDC database when you are logged in as gpadmin. Make sure that the Kerberos `kadm.acl` configuration file contains an ACL to grant permissions to this principal. For example, this ACL grants all permissions to any admin user in the GPDB.KRB realm.

   ```
   */admin@GPDB.KRB *
   ```

4. Create a keytab file with `kadmin.local`. The following example creates a keytab file `gpdb-kerberos.keytab` in the current directory with authentication information for the Greenplum Database service principal and the gpadmin/admin principal.

```
# kadmin.local -q "ktadd -k gpdb-kerberos.keytab postgres/mdw@GPDB.KRB
 gadmin/admin@GPDB.KRB"
```

5. Copy the keytab file to the master host.

```
# scp gpdb-kerberos.keytab gpadmin@mdw:~
```

Installing the Kerberos Client on the Master Host

Install the Kerberos client utilities and libraries on the Greenplum Database master.

1. Install the Kerberos packages on the Greenplum Database master.

```
$ sudo yum install krb5-libs krb5-workstation
```

2. Copy the `/etc/krb5.conf` file from the KDC server to `/etc/krb5.conf` on the Greenplum Master host.

Configuring Greenplum Database to use Kerberos Authentication

Configure Greenplum Database to use Kerberos.

1. Log in to the Greenplum Database master host as the gpadmin user.

```
$ ssh gpadmin@<master>
$ source /usr/local/greenplum-db/greenplum_path.sh
```

2. Set the ownership and permissions of the keytab file you copied from the KDC server.

```
$ chown gpadmin:gpadmin /home/gpadmin/gpdb-kerberos.keytab
$ chmod 400 /home/gpadmin/gpdb-kerberos.keytab
```

3. Configure the location of the keytab file by setting the Greenplum Database `krb_server_keyfile` server configuration parameter. This `gpconfig` command specifies the folder `/home/gpadmin` as the location of the keytab file `gpdb-kerberos.keytab`.

```
$ gpconfig -c krb_server_keyfile -v  '/home/gpadmin/gpdb-kerberos.keytab'
```

4. Modify the Greenplum Database file `pg_hba.conf` to enable Kerberos support. For example, adding the following line to `pg_hba.conf` adds GSSAPI and Kerberos authentication support for connection requests from all users and hosts on the same network to all Greenplum Database databases.

```
host all all 0.0.0.0/0 gss include_realm=0 krb_realm=GPDB.KRB
```

   Setting the `krb_realm` option to a realm name ensures that only users from that realm can successfully authenticate with Kerberos. Setting the `include_realm` option to `0` excludes the realm name from the authenticated user name. For information about the `pg_hba.conf` file, see *The pg_hba.conf file* in the PostgreSQL documentation.

5. Restart Greenplum Database after updating the `krb_server_keyfile` parameter and the `pg_hba.conf` file.

```
$ gpstop -ar
```

6. Create the gpadmin/admin Greenplum Database superuser role.

```
$ createuser gpadmin/admin
```

```
Shall the new role be a superuser? (y/n) y
```

The Kerberos keys for this database role are in the keyfile you copied from the KDC server.

7. Create a ticket using `kinit` and show the tickets in the Kerberos ticket cache with `klist`.

```
$ LD_LIBRARY_PATH= kinit -k -t /home/gpadmin/gpdb-kerberos.keytab gpadmin/
admin@GPDB.KRB
$ LD_LIBRARY_PATH= klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: gpadmin/admin@GPDB.KRB

Valid starting       Expires              Service principal
06/13/2018 17:37:35  06/14/2018 17:37:35  krbtgt/GPDB.KRB@GPDB.KRB
```

Note: When you set up the Greenplum Database environment by sourcing the `greenplum-db_path.sh` script, the `LD_LIBRARY_PATH` environment variable is set to include the Greenplum Database `lib` directory, which includes Kerberos libraries. This may cause Kerberos utility commands such as `kinit` and `klist` to fail due to version conflicts. The solution is to run Kerberos utilities before you source the `greenplum-db_path.sh` file or temporarily unset the `LD_LIBRARY_PATH` variable when you execute Kerberos utilities, as shown in the example.

8. As a test, log in to the postgres database with the `gpadmin/admin` role:

```
$ psql -U "gpadmin/admin" -h mdw postgres
psql (9.4.20)
Type "help" for help.

postgres=# select current_user;
 current_user
---------------
 gpadmin/admin
(1 row)
```

Note: When you start `psql` on the master host, you must include the `-h <master-hostname>` option to force a TCP connection because Kerberos authentication does not work with local connections.

If a Kerberos principal is not a Greenplum Database user, a message similar to the following is displayed from the `psql` command line when the user attempts to log in to the database:

```
psql: krb5_sendauth: Bad response
```

The principal must be added as a Greenplum Database user.

Mapping Kerberos Principals to Greenplum Database Roles

To connect to a Greenplum Database system with Kerberos authentication enabled, a user first requests a ticket-granting ticket from the KDC server using the `kinit` utility with a password or a keytab file provided by the Kerberos admin. When the user then connects to the Kerberos-enabled Greenplum Database system, the user's Kerberos principle name will be the Greenplum Database role name, subject to transformations specified in the options field of the `gss` entry in the Greenplum Database `pg_hba.conf` file:

- If the `krb_realm=<realm>` option is present, Greenplum Database only accepts Kerberos principals who are members pf the specified realm.
- If the `include_realm=0` option is specified, the Greenplum Database role name is the Kerberos principal name without the Kerberos realm. If the `include_realm=1` option is instead specified, the Kerberos realm is not stripped from the Greenplum Database rolename. The role must have been created with the Greenplum Database `CREATE ROLE` command.

- If the `map=<map-name>` option is specified, the Kerberos principal name is compared to entries labeled with the specified `<map-name>` in the `$MASTER_DATA_DIRECTORY/pg_ident.conf` file and replaced with the Greenplum Database role name specified in the first matching entry.

A user name map is defined in the `$MASTER_DATA_DIRECTORY/pg_ident.conf` configuration file. This example defines a map named `mymap` with two entries.

```
# MAPNAME   SYSTEM-USERNAME         GP-USERNAME
mymap       /^admin@GPDB.KRB$       gpadmin
mymap       /^(.*)_gp)@GPDB.KRB$    \1
```

The map name is specified in the `pg_hba.conf` Kerberos entry in the options field:

```
host all all 0.0.0.0/0 gss include_realm=0 krb_realm=GPDB.KRB map=mymap
```

The first map entry matches the Kerberos principal admin@GPDB.KRB and replaces it with the Greenplum Database gpadmin role name. The second entry uses a wildcard to match any Kerberos principal in the GPDB-KRB realm with a name ending with the characters `_gp` and replaces it with the initial portion of the principal name. Greenplum Database applies the first matching map entry in the `pg_ident.conf` file, so the order of entries is significant.

For more information about using username maps see *Username maps* in the PostgreSQL documentation.

Configuring JDBC Kerberos Authentication for Greenplum Database

Enable Kerberos-authenticated JDBC access to Greenplum Database.

You can configure Greenplum Database to use Kerberos to run user-defined Java functions.

1. Ensure that Kerberos is installed and configured on the Greenplum Database master. See *Installing the Kerberos Client on the Master Host*.
2. Create the file `.java.login.config` in the folder `/home/gpadmin` and add the following text to the file:

   ```
   pgjdbc {
     com.sun.security.auth.module.Krb5LoginModule required
     doNotPrompt=true
     useTicketCache=true
     debug=true
     client=true;
   };
   ```
3. Create a Java application that connects to Greenplum Database using Kerberos authentication. The following example database connection URL uses a PostgreSQL JDBC driver and specifies parameters for Kerberos authentication:

   ```
   jdbc:postgresql://mdw:5432/mytest?kerberosServerName=postgres
   &jaasApplicationName=pgjdbc&user=gpadmin/gpdb-kdc
   ```

   The parameter names and values specified depend on how the Java application performs Kerberos authentication.
4. Test the Kerberos login by running a sample Java application from Greenplum Database.

Installing and Configuring a Kerberos KDC Server

Steps to set up a Kerberos Key Distribution Center (KDC) server on a Red Hat Enterprise Linux host for use with Greenplum Database.

If you do not already have a KDC, follow these steps to install and configure a KDC server on a Red Hat Enterprise Linux host with a `GPDB.KRB` realm. The host name of the KDC server in this example is gpdb-kdc.

1. Install the Kerberos server and client packages:

```
$ sudo yum install krb5-libs krb5-server krb5-workstation
```

2. Edit the `/etc/krb5.conf` configuration file. The following example shows a Kerberos server configured with a default `GPDB.KRB` realm.

```
[logging]
 default = FILE:/var/log/krb5libs.log
 kdc = FILE:/var/log/krb5kdc.log
 admin_server = FILE:/var/log/kadmind.log

[libdefaults]
 default_realm = GPDB.KRB
 dns_lookup_realm = false
 dns_lookup_kdc = false
 ticket_lifetime = 24h
 renew_lifetime = 7d
 forwardable = true
 default_tgs_enctypes = aes128-cts des3-hmac-sha1 des-cbc-crc des-cbc-md5
 default_tkt_enctypes = aes128-cts des3-hmac-sha1 des-cbc-crc des-cbc-md5
 permitted_enctypes = aes128-cts des3-hmac-sha1 des-cbc-crc des-cbc-md5

[realms]
 GPDB.KRB = {
  kdc = gpdb-kdc:88
  admin_server = gpdb-kdc:749
  default_domain = gpdb.krb
 }

[domain_realm]
 .gpdb.krb = GPDB.KRB
 gpdb.krb = GPDB.KRB

[appdefaults]
 pam = {
    debug = false
    ticket_lifetime = 36000
    renew_lifetime = 36000
    forwardable = true
    krb4_convert = false
 }
```

   The `kdc` and `admin_server` keys in the `[realms]` section specify the host (`gpdb-kdc`) and port where the Kerberos server is running. IP numbers can be used in place of host names.

   If your Kerberos server manages authentication for other realms, you would instead add the `GPDB.KRB` realm in the `[realms]` and `[domain_realm]` section of the `kdc.conf` file. See the *Kerberos documentation* for information about the `kdc.conf` file.

3. To create the Kerberos database, run the `kdb5_util`.

```
# kdb5_util create -s
```

   The `kdb5_util` create command creates the database to store keys for the Kerberos realms that are managed by this KDC server. The -s option creates a stash file. Without the stash file, every time the KDC server starts it requests a password.

4. Add an administrative user to the KDC database with the `kadmin.local` utility. Because it does not itself depend on Kerberos authentication, the `kadmin.local` utility allows you to add an initial

administrative user to the local Kerberos server. To add the user `gpadmin` as an administrative user to the KDC database, run the following command:

```
# kadmin.local -q "addprinc gpadmin/admin"
```

Most users do not need administrative access to the Kerberos server. They can use `kadmin` to manage their own principals (for example, to change their own password). For information about `kadmin`, see the *Kerberos documentation*.

5. If needed, edit the `/var/kerberos/krb5kdc/kadm5.acl` file to grant the appropriate permissions to `gpadmin`.
6. Start the Kerberos daemons:

```
# /sbin/service krb5kdc start#
/sbin/service kadmin start
```

7. To start Kerberos automatically upon restart:

```
# /sbin/chkconfig krb5kdc on
# /sbin/chkconfig kadmin on
```

Configuring Kerberos for Linux Clients

You can configure Linux client applications to connect to a Greenplum Database system that is configured to authenticate with Kerberos.

If your JDBC application on Red Hat Enterprise Linux uses Kerberos authentication when it connects to your Greenplum Database, your client system must be configured to use Kerberos authentication. If you are not using Kerberos authentication to connect to a Greenplum Database, Kerberos is not needed on your client system.

- *Requirements*
- *Setting Up Client System with Kerberos Authentication*
- *Running a Java Application*

For information about enabling Kerberos authentication with Greenplum Database, see the chapter "Setting Up Kerberos Authentication" in the *Greenplum Database Administrator Guide*.

Requirements

The following are requirements to connect to a Greenplum Database that is enabled with Kerberos authentication from a client system with a JDBC application.

- *Prerequisites*
- *Required Software on the Client Machine*

Prerequisites

- Kerberos must be installed and configured on the Greenplum Database master host.

  Important: Greenplum Database must be configured so that a remote user can connect to Greenplum Database with Kerberos authentication. Authorization to access Greenplum Database is controlled by the `pg_hba.conf` file. For details, see "Editing the pg_hba.conf File" in the *Greenplum Database Administration Guide*, and also see the *Greenplum Database Security Configuration Guide*.
- The client system requires the Kerberos configuration file `krb5.conf` from the Greenplum Database master.
- The client system requires a Kerberos keytab file that contains the authentication credentials for the Greenplum Database user that is used to log into the database.
- The client machine must be able to connect to Greenplum Database master host.

If necessary, add the Greenplum Database master host name and IP address to the system `hosts` file. On Linux systems, the `hosts` file is in `/etc`.

Required Software on the Client Machine

- The Kerberos `kinit` utility is required on the client machine. The `kinit` utility is available when you install the Kerberos packages:
  - krb5-libs
  - krb5-workstation

    Note: When you install the Kerberos packages, you can use other Kerberos utilities such as `klist` to display Kerberos ticket information.

Java applications require this additional software:

- Java JDK

  Java JDK 1.7.0_17 is supported on Red Hat Enterprise Linux 6.x.
- Ensure that JAVA_HOME is set to the installation directory of the supported Java JDK.

Setting Up Client System with Kerberos Authentication

To connect to Greenplum Database with Kerberos authentication requires a Kerberos ticket. On client systems, tickets are generated from Kerberos keytab files with the `kinit` utility and are stored in a cache file.

1. Install a copy of the Kerberos configuration file `krb5.conf` from the Greenplum Database master. The file is used by the Greenplum Database client software and the Kerberos utilities.

   Install `krb5.conf` in the directory `/etc`.

   If needed, add the parameter `default_ccache_name` to the `[libdefaults]` section of the `krb5.ini` file and specify location of the Kerberos ticket cache file on the client system.
2. Obtain a Kerberos keytab file that contains the authentication credentials for the Greenplum Database user.
3. Run `kinit` specifying the keytab file to create a ticket on the client machine. For this example, the keytab file `gpdb-kerberos.keytab` is in the current directory. The ticket cache file is in the `gpadmin` user home directory.

```
> kinit -k -t gpdb-kerberos.keytab -c /home/gpadmin/cache.txt
   gpadmin/kerberos-gpdb@KRB.EXAMPLE.COM
```

Running psql

From a remote system, you can access a Greenplum Database that has Kerberos authentication enabled.

To connect to Greenplum Database with psql

1. As the gpadmin user, open a command window.
2. Start `psql` from the command window and specify a connection to the Greenplum Database specifying the user that is configured with Kerberos authentication.

   The following example logs into the Greenplum Database on the machine `kerberos-gpdb` as the `gpadmin` user with the Kerberos credentials `gpadmin/kerberos-gpdb`:

```
$ psql -U "gpadmin/kerberos-gpdb" -h kerberos-gpdb postgres
```

Running a Java Application

Accessing Greenplum Database from a Java application with Kerberos authentication uses the Java Authentication and Authorization Service (JAAS)

1. Create the file `.java.login.config` in the user home folder.

   For example, on a Linux system, the home folder is similar to `/home/gpadmin`.

   Add the following text to the file:

   ```
   pgjdbc {
     com.sun.security.auth.module.Krb5LoginModule required
     doNotPrompt=true
     useTicketCache=true
     ticketCache = "/home/gpadmin/cache.txt"
     debug=true
     client=true;
   };
   ```

2. Create a Java application that connects to Greenplum Database using Kerberos authentication and run the application as the user.

This example database connection URL uses a PostgreSQL JDBC driver and specifies parameters for Kerberos authentication.

```
jdbc:postgresql://kerberos-gpdb:5432/mytest?
  kerberosServerName=postgres&jaasApplicationName=pgjdbc&
  user=gpadmin/kerberos-gpdb
```

The parameter names and values specified depend on how the Java application performs Kerberos authentication.

Configuring Kerberos For Windows Clients

You can configure Microsoft Windows client applications to connect to a Greenplum Database system that is configured to authenticate with Kerberos.

When a Greenplum Database system is configured to authenticate with Kerberos, you can configure Kerberos authentication for the Greenplum Database client utilities `gpload` and `psql` on a Microsoft Windows system. The Greenplum Database clients authenticate with Kerberos directly.

This section contains the following information.

- *Installing and Configuring Kerberos on a Windows System*
- *Running the psql Utility*
- *Example gpload YAML File*
- *Creating a Kerberos Keytab File*
- *Issues and Possible Solutions*

These topics assume that the Greenplum Database system is configured to authenticate with Kerberos. For information about configuring Greenplum Database with Kerberos authentication, refer to *Using Kerberos Authentication*.

Installing and Configuring Kerberos on a Windows System

The `kinit`, `kdestroy`, and `klist` MIT Kerberos Windows client programs and supporting libraries are installed on your system when you install the Greenplum Database Client and Load Tools package:

- `kinit` - generate a Kerberos ticket
- `kdestroy` - destroy active Kerberos tickets

- `klist` - list Kerberos tickets

You must configure Kerberos on the Windows client to authenticate with Greenplum Database:

1. Copy the Kerberos configuration file `/etc/krb5.conf` from the Greenplum Database master to the Windows system, rename it to `krb5.ini`, and place it in the default Kerberos location on the Windows system, `C:\ProgramData\MIT\Kerberos5\krb5.ini`. This directory may be hidden. This step requires administrative privileges on the Windows client system. You may also choose to place the `/etc/krb5.ini` file in a custom location. If you choose to do this, you must configure and set a system environment variable named `KRB5_CONFIG` to the custom location.
2. Locate the `[libdefaults]` section of the `krb5.ini` file, and remove the entry identifying the location of the Kerberos credentials cache file, `default_ccache_name`. This step requires administrative privileges on the Windows client system.

   This is an example configuration file with `default_ccache_name` removed. The `[logging]` section is also removed.

   ```
   [libdefaults]
    debug = true
    default_etypes = aes256-cts-hmac-sha1-96
    default_realm = EXAMPLE.LOCAL
    dns_lookup_realm = false
    dns_lookup_kdc = false
    ticket_lifetime = 24h
    renew_lifetime = 7d
    forwardable = true

   [realms]
    EXAMPLE.LOCAL = {
     kdc =bocdc.example.local
     admin_server = bocdc.example.local
    }

   [domain_realm]
    .example.local = EXAMPLE.LOCAL
    example.local = EXAMPLE.LOCAL
   ```
3. Set up the Kerberos credential cache file. On the Windows system, set the environment variable `KRB5CCNAME` to specify the file system location of the cache file. The file must be named `krb5cache`. This location identifies a file, not a directory, and should be unique to each login on the server. When you set `KRB5CCNAME`, you can specify the value in either a local user environment or within a session. For example, the following command sets `KRB5CCNAME` in the session:

   ```
   set KRB5CCNAME=%USERPROFILE%\krb5cache
   ```
4. Obtain your Kerberos principal and password or keytab file from your system administrator.
5. Generate a Kerberos ticket using a password or a keytab. For example, to generate a ticket using a password:

   ```
   kinit [<principal>]
   ```

   To generate a ticket using a keytab (as described in *Creating a Kerberos Keytab File*):

   ```
   kinit -k -t <keytab_filepath> [<principal>]
   ```
6. Set up the Greenplum clients environment:

   ```
   set PGGSSLIB=gssapi
   "c:\Program Files\Greenplum\greenplum-clients\greenplum_clients_path.bat"
   ```

Running the psql Utility

After you configure Kerberos and generate the Kerberos ticket on a Windows system, you can run the Greenplum Database command line client `psql`.

If you get warnings indicating that the Console code page differs from Windows code page, you can run the Windows utility `chcp` to change the code page. This is an example of the warning and fix:

```
psql -h prod1.example.local warehouse
psql (9.4.20)
WARNING: Console code page (850) differs from Windows code page (1252)
 8-bit characters might not work correctly. See psql reference
 page "Notes for Windows users" for details.
Type "help" for help.

warehouse=# \q

chcp 1252
Active code page: 1252

psql -h prod1.example.local warehouse
psql (9.4.20)
Type "help" for help.
```

Creating a Kerberos Keytab File

You can create and use a Kerberos `keytab` file to avoid entering a password at the command line or listing a password in a script file when you connect to a Greenplum Database system, perhaps when automating a scheduled Greenplum task such as `gpload`. You can create a keytab file with the Java JRE keytab utility `ktab`. If you use AES256-CTS-HMAC-SHA1-96 encryption, you need to download and install the Java extension *Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files for JDK/JRE* from Oracle.

> **Note:** You must enter the password to create a keytab file. The password is visible onscreen as you enter it.

This example runs the Java `ktab.exe` program to create a keytab file (`-a` option) and list the keytab name and entries (`-l -e -t` options).

```
C:\Users\dev1>"\Program Files\Java\jre1.8.0_77\bin"\ktab -a dev1
Password for dev1@EXAMPLE.LOCAL:your_password
Done!
Service key for dev1 is saved in C:\Users\dev1\krb5.keytab

C:\Users\dev1>"\Program Files\Java\jre1.8.0_77\bin"\ktab -l -e -t
Keytab name: C:\Users\dev1\krb5.keytab
KVNO Timestamp Principal
---- ------------- ------------------------------------------------------
 4 13/04/16 19:14 dev1@EXAMPLE.LOCAL (18:AES256 CTS mode with HMAC SHA1-96)
 4 13/04/16 19:14 dev1@EXAMPLE.LOCAL (17:AES128 CTS mode with HMAC SHA1-96)
 4 13/04/16 19:14 dev1@EXAMPLE.LOCAL (16:DES3 CBC mode with SHA1-KD)
 4 13/04/16 19:14 dev1@EXAMPLE.LOCAL (23:RC4 with HMAC)
```

You can then generate a Kerberos ticket using a keytab with the following command:

```
kinit -kt dev1.keytab dev1
```

or

```
kinit -kt %USERPROFILE%\krb5.keytab dev1
```

Example gpload YAML File

When you initiate a `gpload` job to a Greenplum Database system using Kerberos authentication, you omit the `USER:` property and value from the YAML control file.

This example `gpload` YAML control file named `test.yaml` does not include a `USER:` entry:

```
---
VERSION: 1.0.0.1
DATABASE: warehouse
HOST: prod1.example.local
PORT: 5432

GPLOAD:
   INPUT:
    - SOURCE:
         PORT_RANGE: [18080,18080]
         FILE:
           - /Users/dev1/Downloads/test.csv
    - FORMAT: text
    - DELIMITER: ','
    - QUOTE: '"'
    - ERROR_LIMIT: 25
    - LOG_ERRORS: true
   OUTPUT:
    - TABLE: public.test
    - MODE: INSERT
   PRELOAD:
    - REUSE_TABLES: true
```

These commands run `kinit` using a keytab file, run `gpload.bat` with the `test.yaml` file, and then display successful `gpload` output.

```
kinit -kt %USERPROFILE%\krb5.keytab dev1

gpload.bat -f test.yaml
2016-04-10 16:54:12|INFO|gpload session started 2016-04-10 16:54:12
2016-04-10 16:54:12|INFO|started gpfdist -p 18080 -P 18080 -f "/Users/dev1/
Downloads/test.csv" -t 30
2016-04-10 16:54:13|INFO|running time: 0.23 seconds
2016-04-10 16:54:13|INFO|rows Inserted = 3
2016-04-10 16:54:13|INFO|rows Updated = 0
2016-04-10 16:54:13|INFO|data formatting errors = 0
2016-04-10 16:54:13|INFO|gpload succeeded
```

Issues and Possible Solutions

- This message indicates that Kerberos cannot find your Kerberos credentials cache file:

  ```
  Credentials cache I/O operation failed XXX
  (Kerberos error 193)
  krb5_cc_default() failed
  ```

  To ensure that Kerberos can find the file, set the environment variable `KRB5CCNAME` and run `kinit`.

  ```
  set KRB5CCNAME=%USERPROFILE%\krb5cache
  kinit
  ```

- This `kinit` message indicates that the `kinit -k -t` command could not find the keytab.

  ```
  kinit: Generic preauthentication failure while getting initial credentials
  ```

  Confirm that the full path and filename for the Kerberos keytab file is correct.

Managing Roles and Privileges

The Greenplum Database authorization mechanism stores roles and permissions to access database objects in the database and is administered using SQL statements or command-line utilities.

Greenplum Database manages database access permissions using *roles*. The concept of roles subsumes the concepts of *users* and *groups*. A role can be a database user, a group, or both. Roles can own database objects (for example, tables) and can assign privileges on those objects to other roles to control access to the objects. Roles can be members of other roles, thus a member role can inherit the object privileges of its parent role.

Every Greenplum Database system contains a set of database roles (users and groups). Those roles are separate from the users and groups managed by the operating system on which the server runs. However, for convenience you may want to maintain a relationship between operating system user names and Greenplum Database role names, since many of the client applications use the current operating system user name as the default.

In Greenplum Database, users log in and connect through the master instance, which then verifies their role and access privileges. The master then issues commands to the segment instances behind the scenes as the currently logged in role.

Roles are defined at the system level, meaning they are valid for all databases in the system.

In order to bootstrap the Greenplum Database system, a freshly initialized system always contains one predefined *superuser* role (also referred to as the system user). This role will have the same name as the operating system user that initialized the Greenplum Database system. Customarily, this role is named `gpadmin`. In order to create more roles you first have to connect as this initial role.

Security Best Practices for Roles and Privileges

- **Secure the gpadmin system user.** Greenplum requires a UNIX user id to install and initialize the Greenplum Database system. This system user is referred to as `gpadmin` in the Greenplum documentation. This `gpadmin` user is the default database superuser in Greenplum Database, as well as the file system owner of the Greenplum installation and its underlying data files. This default administrator account is fundamental to the design of Greenplum Database. The system cannot run without it, and there is no way to limit the access of this gpadmin user id. Use roles to manage who has access to the database for specific purposes. You should only use the `gpadmin` account for system maintenance tasks such as expansion and upgrade. Anyone who logs on to a Greenplum host as this user id can read, alter or delete any data; including system catalog data and database access rights. Therefore, it is very important to secure the gpadmin user id and only provide access to essential system administrators. Administrators should only log in to Greenplum as `gpadmin` when performing certain system maintenance tasks (such as upgrade or expansion). Database users should never log on as `gpadmin`, and ETL or production workloads should never run as `gpadmin`.
- **Assign a distinct role to each user that logs in.** For logging and auditing purposes, each user that is allowed to log in to Greenplum Database should be given their own database role. For applications or web services, consider creating a distinct role for each application or service. See *Creating New Roles (Users)*.
- **Use groups to manage access privileges.** See *Role Membership*.
- **Limit users who have the SUPERUSER role attribute.** Roles that are superusers bypass all access privilege checks in Greenplum Database, as well as resource queuing. Only system administrators should be given superuser rights. See *Altering Role Attributes*.

Creating New Roles (Users)

A user-level role is considered to be a database role that can log in to the database and initiate a database session. Therefore, when you create a new user-level role using the `CREATE ROLE` command, you must specify the `LOGIN` privilege. For example:

```
=# CREATE ROLE jsmith WITH LOGIN;
```

A database role may have a number of attributes that define what sort of tasks that role can perform in the database. You can set these attributes when you create the role, or later using the `ALTER ROLE` command. See *Table 37: Role Attributes* for a description of the role attributes you can set.

Altering Role Attributes

A database role may have a number of attributes that define what sort of tasks that role can perform in the database.

Table 37: Role Attributes

Attributes	Description
`SUPERUSER | NOSUPERUSER`	Determines if the role is a superuser. You must yourself be a superuser to create a new superuser. `NOSUPERUSER` is the default.
`CREATEDB | NOCREATEDB`	Determines if the role is allowed to create databases. `NOCREATEDB` is the default.
`CREATEROLE | NOCREATEROLE`	Determines if the role is allowed to create and manage other roles. `NOCREATEROLE` is the default.
`INHERIT | NOINHERIT`	Determines whether a role inherits the privileges of roles it is a member of. A role with the `INHERIT` attribute can automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of. `INHERIT` is the default.
`LOGIN | NOLOGIN`	Determines whether a role is allowed to log in. A role having the `LOGIN` attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges (groups). `NOLOGIN` is the default.
`CONNECTION LIMIT` *`connlimit`*	If role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.
`CREATEEXTTABLE | NOCREATEEXTTABLE`	Determines whether a role is allowed to create external tables. `NOCREATEEXTTABLE` is the default. For a role with the `CREATEEXTTABLE` attribute, the default external table `type` is `readable` and the default `protocol` is `gpfdist`. Note that external tables that use the `file` or `execute` protocols can only be created by superusers.
`PASSWORD '`*`password`*`'`	Sets the role's password. If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as `PASSWORD NULL`.

Attributes	Description
`ENCRYPTED | UNENCRYPTED`	Controls whether a new password is stored as a hash string in the `pg_authid` system catalog. If neither `ENCRYPTED` nor `UNENCRYPTED` is specified, the default behavior is determined by the `password_encryption` configuration parameter, which is `on` by default. If the supplied *`password`* string is already in hashed format, it is stored as-is, regardless of whether `ENCRYPTED` or `UNENCRYPTED` is specified. See *Protecting Passwords in Greenplum Database* for additional information about protecting login passwords.
`VALID UNTIL '`*`timestamp`*`'`	Sets a date and time after which the role's password is no longer valid. If omitted the password will be valid for all time.
`RESOURCE QUEUE` *`queue_name`*	Assigns the role to the named resource queue for workload management. Any statement that role issues is then subject to the resource queue's limits. Note that the `RESOURCE QUEUE` attribute is not inherited; it must be set on each user-level (`LOGIN`) role.
`DENY {deny_interval | deny_point}`	Restricts access during an interval, specified by day or day and time. For more information see *Time-based Authentication*.

You can set these attributes when you create the role, or later using the `ALTER ROLE` command. For example:

```
=# ALTER ROLE jsmith WITH PASSWORD 'passwd123';
=# ALTER ROLE admin VALID UNTIL 'infinity';
=# ALTER ROLE jsmith LOGIN;
=# ALTER ROLE jsmith RESOURCE QUEUE adhoc;
=# ALTER ROLE jsmith DENY DAY 'Sunday';
```

A role can also have role-specific defaults for many of the server configuration settings. For example, to set the default schema search path for a role:

```
=# ALTER ROLE admin SET search_path TO myschema, public;
```

Role Membership

It is frequently convenient to group users together to ease management of object privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In Greenplum Database this is done by creating a role that represents the group, and then granting membership in the group role to individual user roles.

Use the `CREATE ROLE` SQL command to create a new group role. For example:

```
=# CREATE ROLE admin CREATEROLE CREATEDB;
```

Once the group role exists, you can add and remove members (user roles) using the `GRANT` and `REVOKE` commands. For example:

```
=# GRANT admin TO john, sally;
=# REVOKE admin FROM bob;
```

For managing object privileges, you would then grant the appropriate permissions to the group-level role only (see *Table 38: Object Privileges*). The member user roles then inherit the object privileges of the group role. For example:

```
=# GRANT ALL ON TABLE mytable TO admin;
=# GRANT ALL ON SCHEMA myschema TO admin;
=# GRANT ALL ON DATABASE mydb TO admin;
```

The role attributes `LOGIN`, `SUPERUSER`, `CREATEDB`, `CREATEROLE`, `CREATEEXTTABLE`, and `RESOURCE QUEUE` are never inherited as ordinary privileges on database objects are. User members must actually `SET ROLE` to a specific role having one of these attributes in order to make use of the attribute. In the above example, we gave `CREATEDB` and `CREATEROLE` to the `admin` role. If `sally` is a member of `admin`, she could issue the following command to assume the role attributes of the parent role:

```
=> SET ROLE admin;
```

Managing Object Privileges

When an object (table, view, sequence, database, function, language, schema, or tablespace) is created, it is assigned an owner. The owner is normally the role that executed the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, privileges must be granted. Greenplum Database supports the following privileges for each object type:

Table 38: Object Privileges

Object Type	Privileges

Tables, Views, Sequences	`SELECT`
	`INSERT`
	`UPDATE`
	`DELETE`
	`RULE`
	`ALL`

External Tables	`SELECT`
	`RULE`
	`ALL`

Databases	`CONNECT`
	`CREATE`
	`TEMPORARY | TEMP`
	`ALL`

Functions	`EXECUTE`
Procedural Languages	`USAGE`

Schemas	`CREATE`
	`USAGE`
	`ALL`

Object Type	Privileges
Custom Protocol	SELECT INSERT UPDATE DELETE RULE ALL

> **Note:** You must grant privileges for each object individually. For example, granting ALL on a database does not grant full access to the objects within that database. It only grants all of the database-level privileges (CONNECT, CREATE, TEMPORARY) to the database itself.

Use the GRANT SQL command to give a specified role privileges on an object. For example, to grant the role named jsmith insert privileges on the table named mytable:

```
=# GRANT INSERT ON mytable TO jsmith;
```

Similarly, to grant jsmith select privileges only to the column named col1 in the table named table2:

```
=# GRANT SELECT (col1) on TABLE table2 TO jsmith;
```

To revoke privileges, use the REVOKE command. For example:

```
=# REVOKE ALL PRIVILEGES ON mytable FROM jsmith;
```

You can also use the DROP OWNED and REASSIGN OWNED commands for managing objects owned by deprecated roles (Note: only an object's owner or a superuser can drop an object or reassign ownership). For example:

```
=# REASSIGN OWNED BY sally TO bob;
=# DROP OWNED BY visitor;
```

Simulating Row Level Access Control

Greenplum Database does not support row-level access or row-level, labeled security. You can simulate row-level access by using views to restrict the rows that are selected. You can simulate row-level labels by adding an extra column to the table to store sensitivity information, and then using views to control row-level access based on this column. You can then grant roles access to the views rather than to the base table.

Encrypting Data

Greenplum Database is installed with an optional module of encryption/decryption functions called pgcrypto. The pgcrypto functions allow database administrators to store certain columns of data in encrypted form. This adds an extra layer of protection for sensitive data, as data stored in Greenplum Database in encrypted form cannot be read by anyone who does not have the encryption key, nor can it be read directly from the disks.

> **Note:** The pgcrypto functions run inside the database server, which means that all the data and passwords move between pgcrypto and the client application in clear-text. For optimal security, consider also using SSL connections between the client and the Greenplum master server.

To use pgcrypto functions, register the pgcrypto extension in each database in which you want to use the functions. For example:

```
$ psql -d testdb -c "CREATE EXTENSION pgcrypto"
```

See *pgcrypto* in the PostgreSQL documentation for more information about individual functions.

Protecting Passwords in Greenplum Database

In its default configuration, Greenplum Database saves MD5 hashes of login users' passwords in the pg_authid system catalog rather than saving clear text passwords. Anyone who is able to view the pg_authid table can see hash strings, but no passwords. This also ensures that passwords are obscured when the database is dumped to backup files.

The hash function executes when the password is set by using any of the following commands:

- CREATE USER *name* WITH ENCRYPTED PASSWORD '*password*'
- CREATE ROLE *name* WITH LOGIN ENCRYPTED PASSWORD '*password*'
- ALTER USER *name* WITH ENCRYPTED PASSWORD '*password*'
- ALTER ROLE *name* WITH ENCRYPTED PASSWORD '*password*'

The ENCRYPTED keyword may be omitted when the password_encryption system configuration parameter is on, which is the default value. The password_encryption configuration parameter determines whether clear text or hashed passwords are saved when the ENCRYPTED or UNENCRYPTED keyword is not present in the command.

> **Note:** The SQL command syntax and password_encryption configuration variable include the term *encrypt*, but the passwords are not technically encrypted. They are *hashed* and therefore cannot be decrypted.

The hash is calculated on the concatenated clear text password and role name. The MD5 hash produces a 32-byte hexadecimal string prefixed with the characters md5. The hashed password is saved in the rolpassword column of the pg_authid system table.

Although it is not recommended, passwords may be saved in clear text in the database by including the UNENCRYPTED keyword in the command or by setting the password_encryption configuration variable to off. Note that changing the configuration value has no effect on existing passwords, only newly created or updated passwords.

To set password_encryption globally, execute these commands in a shell as the gpadmin user:

```
$ gpconfig -c password_encryption -v 'off'
$ gpstop -u
```

To set password_encryption in a session, use the SQL SET command:

```
=# SET password_encryption = 'on';
```

Passwords may be hashed using the SHA-256 hash algorithm instead of the default MD5 hash algorithm. The algorithm produces a 64-byte hexadecimal string prefixed with the characters sha256.

> **Note:**
>
> Although SHA-256 uses a stronger cryptographic algorithm and produces a longer hash string, it cannot be used with the MD5 authentication method. To use SHA-256 password hashing the authentication method must be set to password in the pg_hba.conf configuration file so that clear text passwords are sent to Greenplum Database. Because clear text passwords are sent over the network, it is very important to use SSL for client connections when you use SHA-256. The default md5 authentication method, on the other hand, hashes the password twice before sending it to Greenplum Database, once on the password and role name and then again with a salt value shared between the client and server, so the clear text password is never sent on the network.

To enable SHA-256 hashing, change the `password_hash_algorithm` configuration parameter from its default value, `md5`, to `sha-256`. The parameter can be set either globally or at the session level. To set `password_hash_algorithm` globally, execute these commands in a shell as the `gpadmin` user:

```
$ gpconfig -c password_hash_algorithm -v 'sha-256'
$ gpstop -u
```

To set `password_hash_algorithm` in a session, use the SQL `SET` command:

```
=# SET password_hash_algorithm = 'sha-256';
```

Time-based Authentication

Greenplum Database enables the administrator to restrict access to certain times by role. Use the `CREATE ROLE` or `ALTER ROLE` commands to specify time-based constraints.

For details, refer to the *Greenplum Database Security Configuration Guide*.

Defining Database Objects

This section covers data definition language (DDL) in Greenplum Database and how to create and manage database objects.

Creating objects in a Greenplum Database includes making up-front choices about data distribution, storage options, data loading, and other Greenplum features that will affect the ongoing performance of your database system. Understanding the options that are available and how the database will be used will help you make the right decisions.

Most of the advanced Greenplum features are enabled with extensions to the SQL `CREATE` DDL statements.

Creating and Managing Databases

A Greenplum Database system is a single instance of Greenplum Database. There can be several separate Greenplum Database systems installed, but usually just one is selected by environment variable settings. See your Greenplum administrator for details.

There can be multiple databases in a Greenplum Database system. This is different from some database management systems (such as Oracle) where the database instance *is* the database. Although you can create many databases in a Greenplum system, client programs can connect to and access only one database at a time — you cannot cross-query between databases.

About Template and Default Databases

Greenplum Database provides some template databases and a default database, *template1*, *template0*, and *postgres*.

By default, each new database you create is based on a *template* database. Greenplum Database uses *template1* to create databases unless you specify another template. Creating objects in *template1* is not recommended. The objects will be in every database you create using the default template database.

Greenplum Database uses another database template, *template0*, internally. Do not drop or modify *template0*. You can use *template0* to create a completely clean database containing only the standard objects predefined by Greenplum Database at initialization.

You can use the *postgres* database to connect to Greenplum Database for the first time. Greenplum Database uses *postgres* as the default database for administrative connections. For example, *postgres* is used by startup processes, the Global Deadlock Detector process, and the FTS (Fault Tolerance Server) process for catalog access.

Creating a Database

The `CREATE DATABASE` command creates a new database. For example:

```
=> CREATE DATABASE new_dbname;
```

To create a database, you must have privileges to create a database or be a Greenplum Database superuser. If you do not have the correct privileges, you cannot create a database. Contact your Greenplum Database administrator to either give you the necessary privilege or to create a database for you.

You can also use the client program `createdb` to create a database. For example, running the following command in a command line terminal connects to Greenplum Database using the provided host name and port and creates a database named *mydatabase*:

```
$ createdb -h masterhost -p 5432 mydatabase
```

The host name and port must match the host name and port of the installed Greenplum Database system.

Some objects, such as roles, are shared by all the databases in a Greenplum Database system. Other objects, such as tables that you create, are known only in the database in which you create them.

Warning: The `CREATE DATABASE` command is not transactional.

Cloning a Database

By default, a new database is created by cloning the standard system database template, *template1*. Any database can be used as a template when creating a new database, thereby providing the capability to 'clone' or copy an existing database and all objects and data within that database. For example:

```
=> CREATE DATABASE new_dbname TEMPLATE old_dbname;
```

Creating a Database with a Different Owner

Another database owner can be assigned when a database is created:

```
=> CREATE DATABASE new_dbname WITH owner=new_user;
```

Viewing the List of Databases

If you are working in the `psql` client program, you can use the `\l` meta-command to show the list of databases and templates in your Greenplum Database system. If using another client program and you are a superuser, you can query the list of databases from the `pg_database` system catalog table. For example:

```
=> SELECT datname from pg_database;
```

Altering a Database

The ALTER DATABASE command changes database attributes such as owner, name, or default configuration attributes. For example, the following command alters a database by setting its default schema search path (the `search_path` configuration parameter):

```
=> ALTER DATABASE mydatabase SET search_path TO myschema, public,
 pg_catalog;
```

To alter a database, you must be the owner of the database or a superuser.

Dropping a Database

The `DROP DATABASE` command drops (or deletes) a database. It removes the system catalog entries for the database and deletes the database directory on disk that contains the data. You must be the database owner or a superuser to drop a database, and you cannot drop a database while you or anyone else is connected to it. Connect to `postgres` (or another database) before dropping a database. For example:

```
=> \c postgres
=> DROP DATABASE mydatabase;
```

You can also use the client program `dropdb` to drop a database. For example, the following command connects to Greenplum Database using the provided host name and port and drops the database *mydatabase*:

```
$ dropdb -h masterhost -p 5432 mydatabase
```

Warning: Dropping a database cannot be undone.

The `DROP DATABASE` command is not transactional.

Creating and Managing Tablespaces

Tablespaces allow database administrators to have multiple file systems per machine and decide how to best use physical storage to store database objects. Tablespaces allow you to assign different storage for frequently and infrequently used database objects or to control the I/O performance on certain database objects. For example, place frequently-used tables on file systems that use high performance solid-state drives (SSD), and place other tables on standard hard drives.

A tablespace requires a host file system location to store its database files. In Greenplum Database, the file system location must exist on all hosts including the hosts running the master, standby master, each primary segment, and each mirror segment.

A tablespace is Greenplum Database system object (a global object), you can use a tablespace from any database if you have appropriate privileges.

Creating a Tablespace

The `CREATE TABLESPACE` command defines a tablespace. For example:

```
CREATE TABLESPACE fastspace LOCATION '/fastdisk/gpdb';
```

Database superusers define tablespaces and grant access to database users with the `GRANTCREATE` command. For example:

```
GRANT CREATE ON TABLESPACE fastspace TO admin;
```

Using a Tablespace to Store Database Objects

Users with the `CREATE` privilege on a tablespace can create database objects in that tablespace, such as tables, indexes, and databases. The command is:

```
CREATE TABLE tablename(options) TABLESPACE spacename
```

For example, the following command creates a table in the tablespace *space1*:

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

You can also use the `default_tablespace` parameter to specify the default tablespace for `CREATE TABLE` and `CREATE INDEX` commands that do not specify a tablespace:

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

There is also the `temp_tablespaces` configuration parameter, which determines the placement of temporary tables and indexes, as well as temporary files that are used for purposes such as sorting large data sets. This can be a comma-separate list of tablespace names, rather than only one, so that the load associated with temporary objects can be spread over multiple tablespaces. A random member of the list is picked each time a temporary object is to be created.

The tablespace associated with a database stores that database's system catalogs, temporary files created by server processes using that database, and is the default tablespace selected for tables and indexes created within the database, if no TABLESPACE is specified when the objects are created. If you do not specify a tablespace when you create a database, the database uses the same tablespace used by its template database.

You can use a tablespace from any database in the Greenplum Database system if you have appropriate privileges.

Viewing Existing Tablespaces

Every Greenplum Database system has the following default tablespaces.

- `pg_global` for shared system catalogs.
- `pg_default`, the default tablespace. Used by the *template1* and *template0* databases.

These tablespaces use the default system location, the data directory locations created at system initialization.

To see tablespace information, use the `pg_tablespace` catalog table to get the objecct ID (OID) of the tablespace and then use `gp_tablespace_location()` function to display the tablespace directories. This is an example that lists one user-defined tablespace, `myspace`:

```
SELECT oid, * FROM pg_tablespace ;

  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
  1664 | pg_global  |       10 |        |
 16391 | myspace    |       10 |        |
(3 rows)
```

The OID for the tablespace `myspace` is `16391`. Run `gp_tablespace_location()` to display the tablespace locations for a system that consists of two segment instances and the master.

```
# SELECT * FROM gp_tablespace_location(16391);
 gp_segment_id |    tblspc_loc
---------------+------------------
             0 | /data/mytblspace
             1 | /data/mytblspace
            -1 | /data/mytblspace
(3 rows)
```

This query uses `gp_tablespace_location()` the `gp_segment_configuration` catalog table to display segment instance information with the file system location for the `myspace` tablespace.

```
WITH spc AS (SELECT * FROM  gp_tablespace_location(16391))
  SELECT seg.role, spc.gp_segment_id as seg_id, seg.hostname, seg.datadir,
 tblspc_loc
    FROM spc, gp_segment_configuration AS seg
    WHERE spc.gp_segment_id = seg.content ORDER BY seg_id;
```

This is information for a test system that consists of two segment instances and the master on a single host.

```
 role | seg_id | hostname |       datadir        |    tblspc_loc
------+--------+----------+----------------------+------------------
 p    |     -1 | testhost | /data/master/gpseg-1 | /data/mytblspace
 p    |      0 | testhost | /data/data1/gpseg0   | /data/mytblspace
 p    |      1 | testhost | /data/data2/gpseg1   | /data/mytblspace
(3 rows)
```

Dropping Tablespaces

To drop a tablespace, you must be the tablespace owner or a superuser. You cannot drop a tablespace until all objects in all databases using the tablespace are removed.

The `DROP TABLESPACE` command removes an empty tablespace.

Note: You cannot drop a tablespace if it is not empty or if it stores temporary or transaction files.

Moving the Location of Temporary or Transaction Files

You can move temporary or transaction files to a specific tablespace to improve database performance when running queries, creating backups, and to store data more sequentially.

The Greenplum Database server configuration parameter `temp_tablespaces` controls the location for both temporary tables and temporary spill files for hash aggregate and hash join queries. (Temporary files for purposes such as sorting large data sets are stored with general segment data in `<data_dir>/<seg_ID>/base/pgsql_tmp`.)

`temp_tablespaces` specifies tablespaces in which to create temporary objects (temp tables and indexes on temp tables) when a `CREATE` command does not explicitly specify a tablespace.

Also note the following information about temporary or transaction files:

- You can dedicate only one tablespace for temporary or transaction files, although you can use the same tablespace to store other types of files.
- You cannot drop a tablespace if it used by temporary files.

Creating and Managing Schemas

Schemas logically organize objects and data in a database. Schemas allow you to have more than one object (such as tables) with the same name in the database without conflict if the objects are in different schemas.

The Default "Public" Schema

Every database has a default schema named *public*. If you do not create any schemas, objects are created in the *public* schema. All database roles (users) have `CREATE` and `USAGE` privileges in the *public* schema. When you create a schema, you grant privileges to your users to allow access to the schema.

Creating a Schema

Use the `CREATE SCHEMA` command to create a new schema. For example:

```
=> CREATE SCHEMA myschema;
```

To create or access objects in a schema, write a qualified name consisting of the schema name and table name separated by a period. For example:

```
myschema.table
```

See *Schema Search Paths* for information about accessing a schema.

You can create a schema owned by someone else, for example, to restrict the activities of your users to well-defined namespaces. The syntax is:

```
=> CREATE SCHEMA schemaname AUTHORIZATION username;
```

Schema Search Paths

To specify an object's location in a database, use the schema-qualified name. For example:

```
=> SELECT * FROM myschema.mytable;
```

You can set the `search_path` configuration parameter to specify the order in which to search the available schemas for objects. The schema listed first in the search path becomes the *default* schema. If a schema is not specified, objects are created in the default schema.

Setting the Schema Search Path

The `search_path` configuration parameter sets the schema search order. The `ALTER DATABASE` command sets the search path. For example:

```
=> ALTER DATABASE mydatabase SET search_path TO myschema,
public, pg_catalog;
```

You can also set `search_path` for a particular role (user) using the `ALTER ROLE` command. For example:

```
=> ALTER ROLE sally SET search_path TO myschema, public,
pg_catalog;
```

Viewing the Current Schema

Use the `current_schema()` function to view the current schema. For example:

```
=> SELECT current_schema();
```

Use the `SHOW` command to view the current search path. For example:

```
=> SHOW search_path;
```

Dropping a Schema

Use the `DROP SCHEMA` command to drop (delete) a schema. For example:

```
=> DROP SCHEMA myschema;
```

By default, the schema must be empty before you can drop it. To drop a schema and all of its objects (tables, data, functions, and so on) use:

```
=> DROP SCHEMA myschema CASCADE;
```

System Schemas

The following system-level schemas exist in every database:

- `pg_catalog` contains the system catalog tables, built-in data types, functions, and operators. It is always part of the schema search path, even if it is not explicitly named in the search path.
- `information_schema` consists of a standardized set of views that contain information about the objects in the database. These views get system information from the system catalog tables in a standardized way.
- `pg_toast` stores large objects such as records that exceed the page size. This schema is used internally by the Greenplum Database system.

- `pg_bitmapindex` stores bitmap index objects such as lists of values. This schema is used internally by the Greenplum Database system.
- `pg_aoseg` stores append-optimized table objects. This schema is used internally by the Greenplum Database system.
- `gp_toolkit` is an administrative schema that contains external tables, views, and functions that you can access with SQL commands. All database users can access `gp_toolkit` to view and query the system log files and other system metrics.

Creating and Managing Tables

Greenplum Database tables are similar to tables in any relational database, except that table rows are distributed across the different segments in the system. When you create a table, you specify the table's distribution policy.

Creating a Table

The `CREATE TABLE` command creates a table and defines its structure. When you create a table, you define:

- The columns of the table and their associated data types. See *Choosing Column Data Types*.
- Any table or column constraints to limit the data that a column or table can contain. See *Setting Table and Column Constraints*.
- The distribution policy of the table, which determines how Greenplum Database divides data across the segments. See *Choosing the Table Distribution Policy*.
- The way the table is stored on disk. See *Choosing the Table Storage Model*.
- The table partitioning strategy for large tables. See *Creating and Managing Databases*.

Choosing Column Data Types

The data type of a column determines the types of data values the column can contain. Choose the data type that uses the least possible space but can still accommodate your data and that best constrains the data. For example, use character data types for strings, date or timestamp data types for dates, and numeric data types for numbers.

For table columns that contain textual data, specify the data type `VARCHAR` or `TEXT`. Specifying the data type `CHAR` is not recommended. In Greenplum Database, the data types `VARCHAR` or `TEXT` handle padding added to the data (space characters added after the last non-space character) as significant characters, the data type `CHAR` does not. For information on the character data types, see the `CREATE TABLE` command in the *Greenplum Database Reference Guide*.

Use the smallest numeric data type that will accommodate your numeric data and allow for future expansion. For example, using `BIGINT` for data that fits in `INT` or `SMALLINT` wastes storage space. If you expect that your data values will expand over time, consider that changing from a smaller datatype to a larger datatype after loading large amounts of data is costly. For example, if your current data values fit in a `SMALLINT` but it is likely that the values will expand, `INT` is the better long-term choice.

Use the same data types for columns that you plan to use in cross-table joins. Cross-table joins usually use the primary key in one table and a foreign key in the other table. When the data types are different, the database must convert one of them so that the data values can be compared correctly, which adds unnecessary overhead.

Greenplum Database has a rich set of native data types available to users. See the *Greenplum Database Reference Guide* for information about the built-in data types.

Setting Table and Column Constraints

You can define constraints on columns and tables to restrict the data in your tables. Greenplum Database support for constraints is the same as PostgreSQL with some limitations, including:

- `CHECK` constraints can refer only to the table on which they are defined.
- `UNIQUE` and `PRIMARY KEY` constraints must be compatible with their table#s distribution key and partitioning key, if any.

  Note: `UNIQUE` and `PRIMARY KEY` constraints are not allowed on append-optimized tables because the `UNIQUE` indexes that are created by the constraints are not allowed on append-optimized tables.
- `FOREIGN KEY` constraints are allowed, but not enforced.
- Constraints that you define on partitioned tables apply to the partitioned table as a whole. You cannot define constraints on the individual parts of the table.

Check Constraints

Check constraints allow you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression. For example, to require positive product prices:

```
=> CREATE TABLE products
           ( product_no integer,
             name text,
             price numeric CHECK (price > 0) );
```

Not-Null Constraints

Not-null constraints specify that a column must not assume the null value. A not-null constraint is always written as a column constraint. For example:

```
=> CREATE TABLE products
       ( product_no integer NOT NULL,
         name text NOT NULL,
         price numeric );
```

Unique Constraints

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table. The table must be hash-distributed or replicated (not `DISTRIBUTED RANDOMLY`). If the table is hash-distributed, the constraint columns must be the same as (or a superset of) the table's distribution key columns. For example:

```
=> CREATE TABLE products
       ( product_no integer UNIQUE,
         name text,
         price numeric)
      DISTRIBUTED BY (product_no);
```

Primary Keys

A primary key constraint is a combination of a `UNIQUE` constraint and a `NOT NULL` constraint. The table must be hash-distributed (not `DISTRIBUTED RANDOMLY`), and the primary key columns must be the same as (or a superset of) the table's distribution key columns. If a table has a primary key, this column (or group of columns) is chosen as the distribution key for the table by default. For example:

```
=> CREATE TABLE products
       ( product_no integer PRIMARY KEY,
         name text,
         price numeric)
      DISTRIBUTED BY (product_no);
```

Foreign Keys

Foreign keys are not supported. You can declare them, but referential integrity is not enforced.

Foreign key constraints specify that the values in a column or a group of columns must match the values appearing in some row of another table to maintain referential integrity between two related tables. Referential integrity checks cannot be enforced between the distributed table segments of a Greenplum database.

Choosing the Table Distribution Policy

All Greenplum Database tables are distributed. When you create or alter a table, you optionally specify `DISTRIBUTED BY` (hash distribution), `DISTRIBUTED RANDOMLY` (round-robin distribution), or `DISTRIBUTED REPLICATED` (fully distributed) to determine the table row distribution.

> **Note:** The Greenplum Database server configuration parameter `gp_create_table_random_default_distribution` controls the table distribution policy if the `DISTRIBUTED BY` clause is not specified when you create a table.
>
> For information about the parameter, see "Server Configuration Parameters" of the *Greenplum Database Reference Guide*.

Consider the following points when deciding on a table distribution policy.

- **Even Data Distribution** — For the best possible performance, all segments should contain equal portions of data. If the data is unbalanced or skewed, the segments with more data must work harder to perform their portion of the query processing. Choose a distribution key that is unique for each record, such as the primary key.
- **Local and Distributed Operations** — Local operations are faster than distributed operations. Query processing is fastest if the work associated with join, sort, or aggregation operations is done locally, at the segment level. Work done at the system level requires distributing tuples across the segments, which is less efficient. When tables share a common distribution key, the work of joining or sorting on their shared distribution key columns is done locally. With a random distribution policy, local join operations are not an option.
- **Even Query Processing** — For best performance, all segments should handle an equal share of the query workload. Query workload can be skewed if a table's data distribution policy and the query predicates are not well matched. For example, suppose that a sales transactions table is distributed on the customer ID column (the distribution key). If a predicate in a query references a single customer ID, the query processing work is concentrated on just one segment.

The replicated table distribution policy (`DISTRIBUTED REPLICATED`) should be used only for small tables. Replicating data to every segment is costly in both storage and maintenance, and prohibitive for large fact tables. The primary use cases for replicated tables are to:

- remove restrictions on operations that user-defined functions can perform on segments, and
- improve query performance by making it unnecessary to broadcast frequently used tables to all segments.

> **Note:** The hidden system columns (`ctid`, `cmin`, `cmax`, `xmin`, `xmax`, and `gp_segment_id`) cannot be referenced in user queries on replicated tables because they have no single, unambiguous value. Greenplum Database returns a `column does not exist` error for the query.

Declaring Distribution Keys

`CREATE TABLE`'s optional clauses `DISTRIBUTED BY`, `DISTRIBUTED RANDOMLY`, and `DISTRIBUTED REPLICATED` specify the distribution policy for a table. The default is a hash distribution policy that uses either the `PRIMARY KEY` (if the table has one) or the first column of the table as the distribution key. Columns with geometric or user-defined data types are not eligible as Greenplum Database distribution

key columns. If a table does not have an eligible column, Greenplum Database distributes the rows randomly or in round-robin fashion.

Replicated tables have no distribution key because every row is distributed to every Greenplum Database segment instance.

To ensure even distribution of hash-distributed data, choose a distribution key that is unique for each record. If that is not possible, choose `DISTRIBUTED RANDOMLY`. For example:

```
=> CREATE TABLE products
                        (name varchar(40),
                         prod_id integer,
                         supplier_id integer)
             DISTRIBUTED BY (prod_id);
```

```
=> CREATE TABLE random_stuff
                        (things text,
                         doodads text,
                         etc text)
             DISTRIBUTED RANDOMLY;
```

> **Important:** If a primary key exists, it is the default distribution key for the table. If no primary key exists, but a unique key exists, this is the default distribution key for the table.

Custom Distribution Key Hash Functions

The hash function used for hash distribution policy is defined by the hash operator class for the column's data type. As the default Greenplum Database uses the data type's default hash operator class, the same operator class used for hash joins and hash aggregates, which is suitable for most use cases. However, you can declare a non-default hash operator class in the `DISTRIBUTED BY` clause.

Using a custom hash operator class can be useful to support co-located joins on a different operator than the default equality operator (=).

Example Custom Hash Operator Class

This example creates a custom hash operator class for the integer data type that is used to improve query performance. The operator class compares the absolute values of integers.

Create a function and an equality operator that returns true if the absolute values of two integers are equal.

```
CREATE FUNCTION abseq(int, int) RETURNS BOOL AS
$$
  begin return abs($1) = abs($2); end;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE OPERATOR |=| (
  PROCEDURE = abseq,
  LEFTARG = int,
  RIGHTARG = int,
  COMMUTATOR = |=|,
  hashes, merges);
```

Now, create a hash function and operator class that uses the operator.

```
CREATE FUNCTION abshashfunc(int) RETURNS int AS
$$
  begin return hashint4(abs($1)); end;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE OPERATOR CLASS abs_int_hash_ops FOR TYPE int4
```

```
  USING hash AS
  OPERATOR 1 |=|,
  FUNCTION 1 abshashfunc(int);
```

Also, create less than and greater than operators, and a btree operator class for them. We don't need them for our queries, but the Postgres Planner will not consider co-location of joins without them.

```
CREATE FUNCTION abslt(int, int) RETURNS BOOL AS
$$
  begin return abs($1) < abs($2); end;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE OPERATOR |<| (
  PROCEDURE = abslt,
  LEFTARG = int,
  RIGHTARG = int);

CREATE FUNCTION absgt(int, int) RETURNS BOOL AS
$$
  begin return abs($1) > abs($2); end;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE OPERATOR |>| (
  PROCEDURE = absgt,
  LEFTARG = int,
  RIGHTARG = int);

CREATE FUNCTION abscmp(int, int) RETURNS int AS
$$
  begin return btint4cmp(abs($1),abs($2)); end;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE OPERATOR CLASS abs_int_btree_ops FOR TYPE int4
  USING btree AS
  OPERATOR 1 |<|,
  OPERATOR 3 |=|,
  OPERATOR 5 |>|,
  FUNCTION 1 abscmp(int, int);
```

Now, you can use the custom hash operator class in tables.

```
CREATE TABLE atab (a int) DISTRIBUTED BY (a abs_int_hash_ops);
CREATE TABLE btab (b int) DISTRIBUTED BY (b abs_int_hash_ops);

INSERT INTO atab VALUES (-1), (0), (1);
INSERT INTO btab VALUES (-1), (0), (1), (2);
```

Queries that perform a join that use the custom equality operator `|=|` can take advantage of the co-location.

With the default integer opclass, this query requires Redistribute Motion nodes.

```
EXPLAIN (COSTS OFF) SELECT a, b FROM atab, btab WHERE a = b;
                            QUERY PLAN
------------------------------------------------------------------
 Gather Motion 4:1  (slice3; segments: 4)
   ->  Hash Join
         Hash Cond: (atab.a = btab.b)
         ->  Redistribute Motion 4:4  (slice1; segments: 4)
               Hash Key: atab.a
               ->  Seq Scan on atab
         ->  Hash
```

```
                ->  Redistribute Motion 4:4  (slice2; segments: 4)
                      Hash Key: btab.b
                      ->  Seq Scan on btab
 Optimizer: Postgres query optimizer
(11 rows)
```

With the custom opclass, a more efficient plan is possible.

```
EXPLAIN (COSTS OFF) SELECT a, b FROM atab, btab WHERE a |=| b;
                            QUERY PLAN
------------------------------------------------------------------
  Gather Motion 4:1  (slice1; segments: 4)
   ->  Hash Join
         Hash Cond: (atab.a |=| btab.b)
         ->  Seq Scan on atab
         ->  Hash
               ->  Seq Scan on btab
 Optimizer: Postgres query optimizer
(7 rows)
```

Choosing the Table Storage Model

Greenplum Database supports several storage models and a mix of storage models. When you create a table, you choose how to store its data. This topic explains the options for table storage and how to choose the best storage model for your workload.

- *Heap Storage*
- *Append-Optimized Storage*
- *Choosing Row or Column-Oriented Storage*
- *Using Compression (Append-Optimized Tables Only)*
- *Checking the Compression and Distribution of an Append-Optimized Table*
- *Altering a Table*
- *Dropping a Table*

> **Note:** To simplify the creation of database tables, you can specify the default values for some table storage options with the Greenplum Database server configuration parameter `gp_default_storage_options`.
>
> For information about the parameter, see "Server Configuration Parameters" in the *Greenplum Database Reference Guide*.

Heap Storage

By default, Greenplum Database uses the same heap storage model as PostgreSQL. Heap table storage works best with OLTP-type workloads where the data is often modified after it is initially loaded. `UPDATE` and `DELETE` operations require storing row-level versioning information to ensure reliable database transaction processing. Heap tables are best suited for smaller tables, such as dimension tables, that are often updated after they are initially loaded.

Append-Optimized Storage

Append-optimized table storage works best with denormalized fact tables in a data warehouse environment. Denormalized fact tables are typically the largest tables in the system. Fact tables are usually loaded in batches and accessed by read-only queries. Moving large fact tables to an append-optimized storage model eliminates the storage overhead of the per-row update visibility information, saving about 20 bytes per row. This allows for a leaner and easier-to-optimize page structure. The storage model of append-optimized tables is optimized for bulk data loading. Single row `INSERT` statements are not recommended.

To create a heap table

Row-oriented heap tables are the default storage type.

```
=> CREATE TABLE foo (a int, b text) DISTRIBUTED BY (a);
```

To create an append-optimized table

Use the `WITH` clause of the `CREATE TABLE` command to declare the table storage options. The default is to create the table as a regular row-oriented heap-storage table. For example, to create an append-optimized table with no compression:

```
=> CREATE TABLE bar (a int, b text)
    WITH (appendoptimized=true)
    DISTRIBUTED BY (a);
```

> **Note:** You use the `appendoptimized=`*`value`* syntax to specify the append-optimized table storage type. `appendoptimized` is a thin alias for the `appendonly` legacy storage option. Greenplum Database stores `appendonly` in the catalog, and displays the same when listing storage options for append-optimized tables.

`UPDATE` and `DELETE` are not allowed on append-optimized tables in a repeatable read or serizalizable transaction and will cause the transaction to abort. `CLUSTER`, `DECLARE...FOR UPDATE`, and triggers are not supported with append-optimized tables.

Choosing Row or Column-Oriented Storage

Greenplum provides a choice of storage orientation models: row, column, or a combination of both. This topic provides general guidelines for choosing the optimum storage orientation for a table. Evaluate performance using your own data and query workloads.

- Row-oriented storage: good for OLTP types of workloads with many iterative transactions and many columns of a single row needed all at once, so retrieving is efficient.
- Column-oriented storage: good for data warehouse workloads with aggregations of data computed over a small number of columns, or for single columns that require regular updates without modifying other column data.

For most general purpose or mixed workloads, row-oriented storage offers the best combination of flexibility and performance. However, there are use cases where a column-oriented storage model provides more efficient I/O and storage. Consider the following requirements when deciding on the storage orientation model for a table:

- **Updates of table data.** If you load and update the table data frequently, choose a row-oriented heap table. Column-oriented table storage is only available on append-optimized tables.

  See *Heap Storage* for more information.
- **Frequent INSERTs.** If rows are frequently inserted into the table, consider a row-oriented model. Column-oriented tables are not optimized for write operations, as column values for a row must be written to different places on disk.
- **Number of columns requested in queries.** If you typically request all or the majority of columns in the `SELECT` list or `WHERE` clause of your queries, consider a row-oriented model. Column-oriented tables are best suited to queries that aggregate many values of a single column where the `WHERE` or `HAVING` predicate is also on the aggregate column. For example:

  ```
  SELECT SUM(salary)...
  ```

  ```
  SELECT AVG(salary)... WHERE salary > 10000
  ```

Or where the `WHERE` predicate is on a single column and returns a relatively small number of rows. For example:

```
SELECT salary, dept ... WHERE state='CA'
```

- **Number of columns in the table.** Row-oriented storage is more efficient when many columns are required at the same time, or when the row-size of a table is relatively small. Column-oriented tables can offer better query performance on tables with many columns where you access a small subset of columns in your queries.
- **Compression.** Column data has the same data type, so storage size optimizations are available in column-oriented data that are not available in row-oriented data. For example, many compression schemes use the similarity of adjacent data to compress. However, the greater adjacent compression achieved, the more difficult random access can become, as data must be uncompressed to be read.

To create a column-oriented table

The `WITH` clause of the `CREATE TABLE` command specifies the table's storage options. The default is a row-oriented heap table. Tables that use column-oriented storage must be append-optimized tables. For example, to create a column-oriented table:

```
=> CREATE TABLE bar (a int, b text)
    WITH (appendoptimized=true, orientation=column)
    DISTRIBUTED BY (a);
```

Using Compression (Append-Optimized Tables Only)

There are two types of in-database compression available in the Greenplum Database for append-optimized tables:

- Table-level compression is applied to an entire table.
- Column-level compression is applied to a specific column. You can apply different column-level compression algorithms to different columns.

The following table summarizes the available compression algorithms.

Table 39: Compression Algorithms for Append-Optimized Tables

Table Orientation	Available Compression Types	Supported Algorithms
Row	Table	`ZLIB`, `ZSTD`, and `QUICKLZ`[1]
Column	Column and Table	`RLE_TYPE`, `ZLIB`, `ZSTD`, and `QUICKLZ`[1]

Note: [1]QuickLZ compression is not available in the open source version of Greenplum Database.

When choosing a compression type and level for append-optimized tables, consider these factors:

- CPU usage. Your segment systems must have the available CPU power to compress and uncompress the data.
- Compression ratio/disk size. Minimizing disk size is one factor, but also consider the time and CPU capacity required to compress and scan data. Find the optimal settings for efficiently compressing data without causing excessively long compression times or slow scan rates.
- Speed of compression. QuickLZ compression generally uses less CPU capacity and compresses data faster at a lower compression ratio than zlib. zlib provides higher compression ratios at lower speeds.

  For example, at compression level 1 (`compresslevel=1`), QuickLZ and zlib have comparable compression ratios, though at different speeds. Using zlib with `compresslevel=6` can significantly increase the compression ratio compared to QuickLZ, though with lower compression speed. Zstandard

compression can provide for either good compression ratio or speed, depending on compression level, or a good compromise on both.

- Speed of decompression/scan rate. Performance with compressed append-optimized tables depends on hardware, query tuning settings, and other factors. Perform comparison testing to determine the actual performance in your environment.

  Note: Do not create compressed append-optimized tables on file systems that use compression. If the file system on which your segment data directory resides is a compressed file system, your append-optimized table must not use compression.

Performance with compressed append-optimized tables depends on hardware, query tuning settings, and other factors. You should perform comparison testing to determine the actual performance in your environment.

Note: Zstd compression level can be set to values between 1 and 19. QuickLZ compression level can only be set to level 1; no other values are available. Compression level with zlib can be set to values from 1 - 9. Compression level with RLE can be set to values from 1 - 4.

An `ENCODING` clause specifies compression type and level for individual columns. When an `ENCODING` clause conflicts with a `WITH` clause, the `ENCODING` clause has higher precedence than the `WITH` clause.

To create a compressed table

The `WITH` clause of the `CREATE TABLE` command declares the table storage options. Tables that use compression must be append-optimized tables. For example, to create an append-optimized table with zlib compression at a compression level of 5:

```
=> CREATE TABLE foo (a int, b text)
   WITH (appendoptimized=true, compresstype=zlib, compresslevel=5);
```

Checking the Compression and Distribution of an Append-Optimized Table

Greenplum provides built-in functions to check the compression ratio and the distribution of an append-optimized table. The functions take either the object ID or a table name. You can qualify the table name with a schema name.

Table 40: Functions for compressed append-optimized table metadata

Function	Return Type	Description
get_ao_distribution(name) get_ao_distribution(oid)	Set of (dbid, tuplecount) rows	Shows the distribution of an append-optimized table's rows across the array. Returns a set of rows, each of which includes a segment *dbid* and the number of tuples stored on the segment.
get_ao_compression_ratio(name) get_ao_compression_ratio(oid)	float8	Calculates the compression ratio for a compressed append-optimized table. If information is not available, this function returns a value of -1.

The compression ratio is returned as a common ratio. For example, a returned value of `3.19`, or `3.19:1`, means that the uncompressed table is slightly larger than three times the size of the compressed table.

The distribution of the table is returned as a set of rows that indicate how many tuples are stored on each segment. For example, in a system with four primary segments with *dbid* values ranging from 0 - 3, the function returns four rows similar to the following:

```
=# SELECT get_ao_distribution('lineitem_comp');
 get_ao_distribution
---------------------
(0,7500721)
(1,7501365)
(2,7499978)
(3,7497731)
(4 rows)
```

Support for Run-length Encoding

Greenplum Database supports Run-length Encoding (RLE) for column-level compression. RLE data compression stores repeated data as a single data value and a count. For example, in a table with two columns, a date and a description, that contains 200,000 entries containing the value `date1` and 400,000 entries containing the value `date2`, RLE compression for the date field is similar to `date1 200000 date2 400000`. RLE is not useful with files that do not have large sets of repeated data as it can greatly increase the file size.

There are four levels of RLE compression available. The levels progressively increase the compression ratio, but decrease the compression speed.

Greenplum Database versions 4.2.1 and later support column-oriented RLE compression. To backup a table with RLE compression that you intend to restore to an earlier version of Greenplum Database, alter the table to have no compression or a compression type supported in the earlier version (`ZLIB` or `QUICKLZ`) before you start the backup operation.

Greenplum Database combines delta compression with RLE compression for data in columns of type `BIGINT`, `INTEGER`, `DATE`, `TIME`, or `TIMESTAMP`. The delta compression algorithm is based on the change between consecutive column values and is designed to improve compression when data is loaded in sorted order or when the compression is applied to data in sorted order.

Adding Column-level Compression

You can add the following storage directives to a column for append-optimized tables with column orientation:

- Compression type
- Compression level
- Block size for a column

Add storage directives using the `CREATE TABLE`, `ALTER TABLE`, and `CREATE TYPE` commands.

The following table details the types of storage directives and possible values for each.

Table 41: Storage Directives for Column-level Compression

Name	Definition	Values	Comment
`compresstype`	Type of compression.	`zstd`: Zstandard algorithm `zlib`: deflate algorithm `quicklz`: fast compression `RLE_TYPE`: run-length encoding `none`: no compression	Values are not case-sensitive.
`compresslevel`	Compression level.	`zlib` compression: `1-9`	`1` is the fastest method with the least compression. `1` is the default. `9` is the slowest method with the most compression.
		`zstd` compression: `1-19`	`1` is the fastest method with the least compression. `1` is the default. `19` is the slowest method with the most compression.
		`QuickLZ` compression: `1` – use compression	`1` is the default.
		`RLE_TYPE` compression: `1` – `4` `1` - apply RLE only `2` - apply RLE then apply zlib compression level 1 `3` - apply RLE then apply zlib compression level 5 `4` - apply RLE then apply zlib compression level 9	`1` is the fastest method with the least compression. `4` is the slowest method with the most compression. `1` is the default.
`blocksize`	The size in bytes for each block in the table	`8192 – 2097152`	The value must be a multiple of 8192.

The following is the format for adding storage directives.

```
[ ENCODING ( storage_directive [,…] ) ]
```

where the word ENCODING is required and the storage directive has three parts:

- The name of the directive

- An equals sign
- The specification

Separate multiple storage directives with a comma. Apply a storage directive to a single column or designate it as the default for all columns, as shown in the following `CREATE TABLE` clauses.

General Usage:

```
column_name data_type ENCODING ( storage_directive [, … ] ), …
```

```
COLUMN column_name ENCODING ( storage_directive [, … ] ), …
```

```
DEFAULT COLUMN ENCODING ( storage_directive [, … ] )
```

Example:

```
C1 char ENCODING (compresstype=quicklz, blocksize=65536)
```

```
COLUMN C1 ENCODING (compresstype=zlib, compresslevel=6, blocksize=65536)
```

```
DEFAULT COLUMN ENCODING (compresstype=quicklz)
```

Default Compression Values

If the compression type, compression level and block size are not defined, the default is no compression, and the block size is set to the Server Configuration Parameter `block_size`.

Precedence of Compression Settings

Column compression settings are inherited from the type level to the table level to the partition level to the subpartition level. The lowest-level settings have priority.

- Column compression settings defined at the table level override any compression settings for the type.
- Column compression settings specified at the table level override any compression settings for the entire table.
- Column compression settings specified for partitions override any compression settings at the column or table levels.
- Column compression settings specified for subpartitions override any compression settings at the partition, column or table levels.
- When an `ENCODING` clause conflicts with a `WITH` clause, the `ENCODING` clause has higher precedence than the `WITH` clause.

  Note: The `INHERITS` clause is not allowed in a table that contains a storage directive or a column reference storage directive.

  Tables created using the `LIKE` clause ignore storage directive and column reference storage directives.

Optimal Location for Column Compression Settings

The best practice is to set the column compression settings at the level where the data resides. See *Example 5*, which shows a table with a partition depth of 2. `RLE_TYPE` compression is added to a column at the subpartition level.

Storage Directives Examples

The following examples show the use of storage directives in `CREATE TABLE` statements.

Example 1

In this example, column `c1` is compressed using `zstd` and uses the block size defined by the system. Column `c2` is compressed with `quicklz`, and uses a block size of `65536`. Column `c3` is not compressed and uses the block size defined by the system.

```
CREATE TABLE T1 (c1 int ENCODING (compresstype=zstd),
                 c2 char ENCODING (compresstype=quicklz, blocksize=65536),
                 c3 char)    WITH (appendoptimized=true,
 orientation=column);
```

Example 2

In this example, column `c1` is compressed using `zlib` and uses the block size defined by the system. Column `c2` is compressed with `quicklz`, and uses a block size of `65536`. Column `c3` is compressed using `RLE_TYPE` and uses the block size defined by the system.

```
CREATE TABLE T2 (c1 int ENCODING (compresstype=zlib),
                 c2 char ENCODING (compresstype=quicklz, blocksize=65536),
                 c3 char,
                 COLUMN c3 ENCODING (compresstype=RLE_TYPE)
                 )
    WITH (appendoptimized=true, orientation=column);
```

Example 3

In this example, column `c1` is compressed using `zlib` and uses the block size defined by the system. Column `c2` is compressed with `quicklz`, and uses a block size of `65536`. Column `c3` is compressed using `zlib` and uses the block size defined by the system. Note that column `c3` uses `zlib` (not `RLE_TYPE`) in the partitions, because the column storage in the partition clause has precedence over the storage directive in the column definition for the table.

```
CREATE TABLE T3 (c1 int ENCODING (compresstype=zlib),
                 c2 char ENCODING (compresstype=quicklz, blocksize=65536),
                 c3 text, COLUMN c3 ENCODING (compresstype=RLE_TYPE) )
    WITH (appendoptimized=true, orientation=column)
    PARTITION BY RANGE (c3) (START ('1900-01-01'::DATE)
                             END ('2100-12-31'::DATE),
                             COLUMN c3 ENCODING (compresstype=zlib));
```

Example 4

In this example, `CREATE TABLE` assigns the `zlib` `compresstype` storage directive to `c1`. Column `c2` has no storage directive and inherits the compression type (`quicklz`) and block size (`65536`) from the `DEFAULT COLUMN ENCODING` clause.

Column `c3`'s `ENCODING` clause defines its compression type, `RLE_TYPE`. The `ENCODING` clause defined for a specific column overrides the `DEFAULT ENCODING` clause, so column `c3` uses the default block size, `32768`.

Column `c4` has a compress type of `none` and uses the default block size.

```
CREATE TABLE T4 (c1 int ENCODING (compresstype=zlib),
                 c2 char,
                 c3 text,
                 c4 smallint ENCODING (compresstype=none),
                 DEFAULT COLUMN ENCODING (compresstype=quicklz,
                                          blocksize=65536),
                 COLUMN c3 ENCODING (compresstype=RLE_TYPE)
```

```
                  )
   WITH (appendoptimized=true, orientation=column);
```

Example 5

This example creates an append-optimized, column-oriented table, T5. T5 has two partitions, `p1` and `p2`, each of which has subpartitions. Each subpartition has `ENCODING` clauses:

- The `ENCODING` clause for partition `p1`'s subpartition `sp1` defines column `i`'s compression type as `zlib` and block size as 65536.
- The `ENCODING` clauses for partition `p2`'s subpartition `sp1` defines column `i`'s compression type as `rle_type` and block size is the default value. Column `k` uses the default compression and its block size is 8192.

```
CREATE TABLE T5(i int, j int, k int, l int)
    WITH (appendoptimized=true, orientation=column)
    PARTITION BY range(i) SUBPARTITION BY range(j)
    (
       partition p1 start(1) end(2)
       ( subpartition sp1 start(1) end(2)
         column i encoding(compresstype=zlib, blocksize=65536)
       ),
       partition p2 start(2) end(3)
       ( subpartition sp1 start(1) end(2)
           column i encoding(compresstype=rle_type)
           column k encoding(blocksize=8192)
       )
    );
```

For an example showing how to add a compressed column to an existing table with the `ALTER TABLE` command, see *Adding a Compressed Column to Table* .

Adding Compression in a TYPE Command

When you create a new type, you can define default compression attributes for the type. For example, the following `CREATE TYPE` command defines a type named `int33` that specifies `quicklz` compression:

```
CREATE TYPE int33 (
   internallength = 4,
   input = int33_in,
   output = int33_out,
   alignment = int4,
   default = 123,
   passedbyvalue,
   compresstype="quicklz",
   blocksize=65536,
   compresslevel=1
   );
```

When you specify `int33` as a column type in a `CREATE TABLE` command, the column is created with the storage directives you specified for the type:

```
CREATE TABLE t2 (c1 int33)
    WITH (appendoptimized=true, orientation=column);
```

Table- or column- level storage attributes that you specify in a table definition override type-level storage attributes. For information about creating and adding compression attributes to a type, see *CREATE TYPE*. For information about changing compression specifications in a type, see *ALTER TYPE*.

Choosing Block Size

The blocksize is the size, in bytes, for each block in a table. Block sizes must be between 8192 and 2097152 bytes, and be a multiple of 8192. The default is 32768.

Specifying large block sizes can consume large amounts of memory. Block size determines buffering in the storage layer. Greenplum maintains a buffer per partition, and per column in column-oriented tables. Tables with many partitions or columns consume large amounts of memory.

Altering a Table

The `ALTER TABLE` command changes the definition of a table. Use `ALTER TABLE` to change table attributes such as column definitions, distribution policy, storage model, and partition structure (see also *Maintaining Partitioned Tables*). For example, to add a not-null constraint to a table column:

```
=> ALTER TABLE address ALTER COLUMN street SET NOT NULL;
```

Altering Table Distribution

`ALTER TABLE` provides options to change a table's distribution policy. When the table distribution options change, the table data may be redistributed on disk, which can be resource intensive. You can also redistribute table data using the existing distribution policy.

Changing the Distribution Policy

For partitioned tables, changes to the distribution policy apply recursively to the child partitions. This operation preserves the ownership and all other attributes of the table. For example, the following command redistributes the table sales across all segments using the customer_id column as the distribution key:

```
ALTER TABLE sales SET DISTRIBUTED BY (customer_id);
```

When you change the hash distribution of a table, table data is automatically redistributed. Changing the distribution policy to a random distribution does not cause the data to be redistributed. For example, the following `ALTER TABLE` command has no immediate effect:

```
ALTER TABLE sales SET DISTRIBUTED RANDOMLY;
```

Changing the distribution policy of a table to `DISTRIBUTED REPLICATED` or from `DISTRIBUTED REPLICATED` automatically redistributes the table data.

Redistributing Table Data

To redistribute table data for tables with a random distribution policy (or when the hash distribution policy has not changed) use `REORGANIZE=TRUE`. Reorganizing data may be necessary to correct a data skew problem, or when segment resources are added to the system. For example, the following command redistributes table data across all segments using the current distribution policy, including random distribution.

```
ALTER TABLE sales SET WITH (REORGANIZE=TRUE);
```

Changing the distribution policy of a table to `DISTRIBUTED REPLICATED` or from `DISTRIBUTED REPLICATED` always redistributes the table data, even when you use `REORGANIZE=FALSE`.

Altering the Table Storage Model

Table storage, compression, and orientation can be declared only at creation. To change the storage model, you must create a table with the correct storage options, load the original table data into the new table, drop the original table, and rename the new table with the original table's name. You must also re-grant any table permissions. For example:

```
CREATE TABLE sales2 (LIKE sales)
WITH (appendoptimized=true, compresstype=quicklz,
      compresslevel=1, orientation=column);
INSERT INTO sales2 SELECT * FROM sales;
DROP TABLE sales;
ALTER TABLE sales2 RENAME TO sales;
GRANT ALL PRIVILEGES ON sales TO admin;
GRANT SELECT ON sales TO guest;
```

See *Splitting a Partition* to learn how to change the storage model of a partitioned table.

Adding a Compressed Column to Table

Use `ALTER TABLE` command to add a compressed column to a table. All of the options and constraints for compressed columns described in *Adding Column-level Compression* apply to columns added with the `ALTER TABLE` command.

The following example shows how to add a column with `zlib` compression to a table, `T1`.

```
ALTER TABLE T1
      ADD COLUMN c4 int DEFAULT 0
      ENCODING (compresstype=zlib);
```

Inheritance of Compression Settings

A partition added to a table that has subpartitions defined with compression settings inherits the compression settings from the subpartition. The following example shows how to create a table with subpartition encodings, then alter it to add a partition.

```
CREATE TABLE ccddl (i int, j int, k int, l int)
  WITH
    (appendoptimized = TRUE, orientation=COLUMN)
  PARTITION BY range(j)
  SUBPARTITION BY list (k)
  SUBPARTITION template(
    SUBPARTITION sp1 values(1, 2, 3, 4, 5),
    COLUMN i ENCODING(compresstype=ZLIB),
    COLUMN j ENCODING(compresstype=QUICKLZ),
    COLUMN k ENCODING(compresstype=ZLIB),
    COLUMN l ENCODING(compresstype=ZLIB))
  (PARTITION p1 START(1) END(10),
   PARTITION p2 START(10) END(20))
;

ALTER TABLE ccddl
  ADD PARTITION p3 START(20) END(30)
;
```

Running the `ALTER TABLE` command creates partitions of table `ccddl` named `ccddl_1_prt_p3` and `ccddl_1_prt_p3_2_prt_sp1`. Partition `ccddl_1_prt_p3` inherits the different compression encodings of subpartition `sp1`.

Dropping a Table

The `DROP TABLE` command removes tables from the database. For example:

```
DROP TABLE mytable;
```

To empty a table of rows without removing the table definition, use `DELETE` or `TRUNCATE`. For example:

```
DELETE FROM mytable;

TRUNCATE mytable;
```

`DROP TABLE` always removes any indexes, rules, triggers, and constraints that exist for the target table. Specify `CASCADE` to drop a table that is referenced by a view. `CASCADE` removes dependent views.

Partitioning Large Tables

Table partitioning enables supporting very large tables, such as fact tables, by logically dividing them into smaller, more manageable pieces. Partitioned tables can improve query performance by allowing the Greenplum Database query optimizer to scan only the data needed to satisfy a given query instead of scanning all the contents of a large table.

- *About Table Partitioning*
- *Deciding on a Table Partitioning Strategy*
- *Creating Partitioned Tables*
- *Loading Partitioned Tables*
- *Verifying Your Partition Strategy*
- *Viewing Your Partition Design*
- *Maintaining Partitioned Tables*

About Table Partitioning

Partitioning does not change the physical distribution of table data across the segments. Table distribution is physical: Greenplum Database physically divides partitioned tables and non-partitioned tables across segments to enable parallel query processing. Table *partitioning* is logical: Greenplum Database logically divides big tables to improve query performance and facilitate data warehouse maintenance tasks, such as rolling old data out of the data warehouse.

Greenplum Database supports:

- *range partitioning*: division of data based on a numerical range, such as date or price.
- *list partitioning*: division of data based on a list of values, such as sales territory or product line.
- A combination of both types.

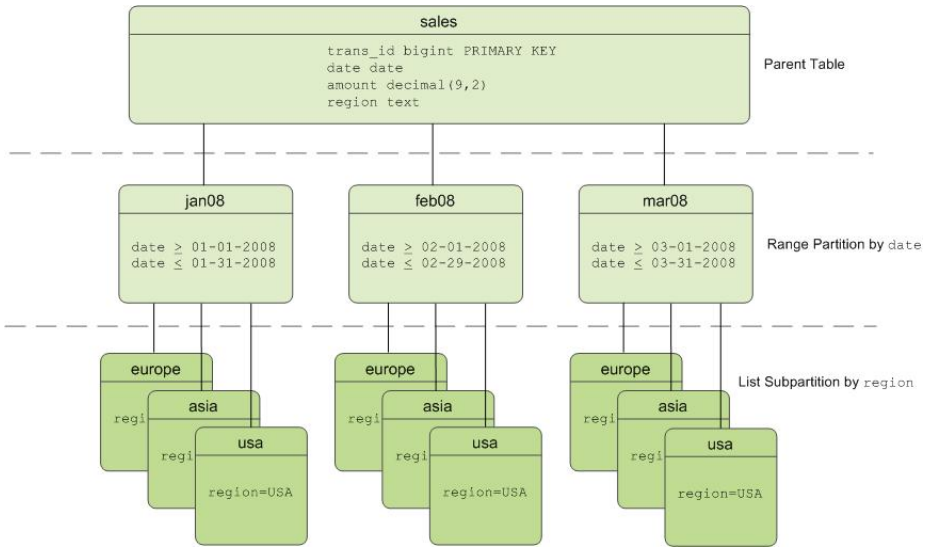


Figure 20: Example Multi-level Partition Design

Table Partitioning in Greenplum Database

Greenplum Database divides tables into parts (also known as partitions) to enable massively parallel processing. Tables are partitioned during `CREATE TABLE` using the `PARTITION BY` (and optionally the `SUBPARTITION BY`) clause. Partitioning creates a top-level (or parent) table with one or more levels of sub-tables (or child tables). Internally, Greenplum Database creates an inheritance relationship between the top-level table and its underlying partitions, similar to the functionality of the `INHERITS` clause of PostgreSQL.

Greenplum uses the partition criteria defined during table creation to create each partition with a distinct `CHECK` constraint, which limits the data that table can contain. The query optimizer uses `CHECK` constraints to determine which table partitions to scan to satisfy a given query predicate.

The Greenplum system catalog stores partition hierarchy information so that rows inserted into the top-level parent table propagate correctly to the child table partitions. To change the partition design or table structure, alter the parent table using `ALTER TABLE` with the `PARTITION` clause.

To insert data into a partitioned table, you specify the root partitioned table, the table created with the `CREATE TABLE` command. You also can specify a leaf child table of the partitioned table in an `INSERT` command. An error is returned if the data is not valid for the specified leaf child table. Specifying a non-leaf or a non-root partition table in the DML command is not supported.

Deciding on a Table Partitioning Strategy

Greenplum Database does not support partitioning replicated tables (`DISTRIBUTED REPLICATED`). Not all hash-distributed or randomly distributed tables are good candidates for partitioning. If the answer is *yes* to all or most of the following questions, table partitioning is a viable database design strategy for improving query performance. If the answer is *no* to most of the following questions, table partitioning is not the right solution for that table. Test your design strategy to ensure that query performance improves as expected.

- **Is the table large enough?** Large fact tables are good candidates for table partitioning. If you have millions or billions of records in a table, you may see performance benefits from logically breaking that data up into smaller chunks. For smaller tables with only a few thousand rows or less, the administrative overhead of maintaining the partitions will outweigh any performance benefits you might see.
- **Are you experiencing unsatisfactory performance?** As with any performance tuning initiative, a table should be partitioned only if queries against that table are producing slower response times than desired.

- **Do your query predicates have identifiable access patterns?** Examine the `WHERE` clauses of your query workload and look for table columns that are consistently used to access data. For example, if most of your queries tend to look up records by date, then a monthly or weekly date-partitioning design might be beneficial. Or if you tend to access records by region, consider a list-partitioning design to divide the table by region.
- **Does your data warehouse maintain a window of historical data?** Another consideration for partition design is your organization's business requirements for maintaining historical data. For example, your data warehouse may require that you keep data for the past twelve months. If the data is partitioned by month, you can easily drop the oldest monthly partition from the warehouse and load current data into the most recent monthly partition.
- **Can the data be divided into somewhat equal parts based on some defining criteria?** Choose partitioning criteria that will divide your data as evenly as possible. If the partitions contain a relatively equal number of records, query performance improves based on the number of partitions created. For example, by dividing a large table into 10 partitions, a query will execute 10 times faster than it would against the unpartitioned table, provided that the partitions are designed to support the query's criteria.

Do not create more partitions than are needed. Creating too many partitions can slow down management and maintenance jobs, such as vacuuming, recovering segments, expanding the cluster, checking disk usage, and others.

Partitioning does not improve query performance unless the query optimizer can eliminate partitions based on the query predicates. Queries that scan every partition run slower than if the table were not partitioned, so avoid partitioning if few of your queries achieve partition elimination. Check the explain plan for queries to make sure that partitions are eliminated. See *Query Profiling* for more about partition elimination.

> **Warning:** Be very careful with multi-level partitioning because the number of partition files can grow very quickly. For example, if a table is partitioned by both day and city, and there are 1,000 days of data and 1,000 cities, the total number of partitions is one million. Column-oriented tables store each column in a physical table, so if this table has 100 columns, the system would be required to manage 100 million files for the table, for each segment.

Before settling on a multi-level partitioning strategy, consider a single level partition with bitmap indexes. Indexes slow down data loads, so performance testing with your data and schema is recommended to decide on the best strategy.

Creating Partitioned Tables

You partition tables when you create them with `CREATE TABLE`. This topic provides examples of SQL syntax for creating a table with various partition designs.

To partition a table:

1. Decide on the partition design: date range, numeric range, or list of values.
2. Choose the column(s) on which to partition the table.
3. Decide how many levels of partitions you want. For example, you can create a date range partition table by month and then subpartition the monthly partitions by sales region.

- *Defining Date Range Table Partitions*
- *Defining Numeric Range Table Partitions*
- *Defining List Table Partitions*
- *Defining Multi-level Partitions*
- *Partitioning an Existing Table*

Defining Date Range Table Partitions

A date range partitioned table uses a single `date` or `timestamp` column as the partition key column. You can use the same partition key column to create subpartitions if necessary, for example, to partition by month and then subpartition by day. Consider partitioning by the most granular level. For example, for a table partitioned by date, you can partition by day and have 365 daily partitions, rather than partition by

year then subpartition by month then subpartition by day. A multi-level design can reduce query planning time, but a flat partition design runs faster.

You can have Greenplum Database automatically generate partitions by giving a `START` value, an `END` value, and an `EVERY` clause that defines the partition increment value. By default, `START` values are always inclusive and `END` values are always exclusive. For example:

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( START (date '2016-01-01') INCLUSIVE
   END (date '2017-01-01') EXCLUSIVE
   EVERY (INTERVAL '1 day') );
```

You can also declare and name each partition individually. For example:

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( PARTITION Jan16 START (date '2016-01-01') INCLUSIVE ,
  PARTITION Feb16 START (date '2016-02-01') INCLUSIVE ,
  PARTITION Mar16 START (date '2016-03-01') INCLUSIVE ,
  PARTITION Apr16 START (date '2016-04-01') INCLUSIVE ,
  PARTITION May16 START (date '2016-05-01') INCLUSIVE ,
  PARTITION Jun16 START (date '2016-06-01') INCLUSIVE ,
  PARTITION Jul16 START (date '2016-07-01') INCLUSIVE ,
  PARTITION Aug16 START (date '2016-08-01') INCLUSIVE ,
  PARTITION Sep16 START (date '2016-09-01') INCLUSIVE ,
  PARTITION Oct16 START (date '2016-10-01') INCLUSIVE ,
  PARTITION Nov16 START (date '2016-11-01') INCLUSIVE ,
  PARTITION Dec16 START (date '2016-12-01') INCLUSIVE
                  END (date '2017-01-01') EXCLUSIVE );
```

You do not have to declare an `END` value for each partition, only the last one. In this example, `Jan16` ends where `Feb16` starts.

Defining Numeric Range Table Partitions

A numeric range partitioned table uses a single numeric data type column as the partition key column. For example:

```
CREATE TABLE rank (id int, rank int, year int, gender
char(1), count int)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
( START (2006) END (2016) EVERY (1),
  DEFAULT PARTITION extra );
```

For more information about default partitions, see *Adding a Default Partition*.

Defining List Table Partitions

A list partitioned table can use any data type column that allows equality comparisons as its partition key column. A list partition can also have a multi-column (composite) partition key, whereas a range partition only allows a single column as the partition key. For list partitions, you must declare a partition specification for every partition (list value) you want to create. For example:

```
CREATE TABLE rank (id int, rank int, year int, gender
char(1), count int )
DISTRIBUTED BY (id)
PARTITION BY LIST (gender)
```

```
( PARTITION girls VALUES ('F'),
  PARTITION boys VALUES ('M'),
  DEFAULT PARTITION other );
```

> **Note:** The current Postgres Planner allows list partitions with multi-column (composite) partition keys. A range partition only allows a single column as the partition key. The Greenplum Query Optimizer does not support composite keys, so you should not use composite partition keys.

For more information about default partitions, see *Adding a Default Partition*.

Defining Multi-level Partitions

You can create a multi-level partition design with subpartitions of partitions. Using a *subpartition template* ensures that every partition has the same subpartition design, including partitions that you add later. For example, the following SQL creates the two-level partition design shown in *Figure 20: Example Multi-level Partition Design*:

```
CREATE TABLE sales (trans_id int, date date, amount
decimal(9,2), region text)
DISTRIBUTED BY (trans_id)
PARTITION BY RANGE (date)
SUBPARTITION BY LIST (region)
SUBPARTITION TEMPLATE
( SUBPARTITION usa VALUES ('usa'),
  SUBPARTITION asia VALUES ('asia'),
  SUBPARTITION europe VALUES ('europe'),
  DEFAULT SUBPARTITION other_regions)
  (START (date '2011-01-01') INCLUSIVE
   END (date '2012-01-01') EXCLUSIVE
   EVERY (INTERVAL '1 month'),
   DEFAULT PARTITION outlying_dates );
```

The following example shows a three-level partition design where the `sales` table is partitioned by `year`, then `month`, then `region`. The `SUBPARTITION TEMPLATE` clauses ensure that each yearly partition has the same subpartition structure. The example declares a `DEFAULT` partition at each level of the hierarchy.

```
CREATE TABLE p3_sales (id int, year int, month int, day int,
region text)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
    SUBPARTITION BY RANGE (month)
       SUBPARTITION TEMPLATE (
        START (1) END (13) EVERY (1),
        DEFAULT SUBPARTITION other_months )
           SUBPARTITION BY LIST (region)
             SUBPARTITION TEMPLATE (
               SUBPARTITION usa VALUES ('usa'),
               SUBPARTITION europe VALUES ('europe'),
               SUBPARTITION asia VALUES ('asia'),
               DEFAULT SUBPARTITION other_regions )
( START (2002) END (2012) EVERY (1),
  DEFAULT PARTITION outlying_years );
```

> **Caution:** When you create multi-level partitions on ranges, it is easy to create a large number of subpartitions, some containing little or no data. This can add many entries to the system tables, which increases the time and memory required to optimize and execute queries. Increase the range interval or choose a different partitioning strategy to reduce the number of subpartitions created.

Partitioning an Existing Table

Tables can be partitioned only at creation. If you have a table that you want to partition, you must create a partitioned table, load the data from the original table into the new table, drop the original table, and rename the partitioned table with the original table's name. You must also re-grant any table permissions. For example:

```
CREATE TABLE sales2 (LIKE sales)
PARTITION BY RANGE (date)
( START (date 2016-01-01') INCLUSIVE
   END (date '2017-01-01') EXCLUSIVE
   EVERY (INTERVAL '1 month') );
INSERT INTO sales2 SELECT * FROM sales;
DROP TABLE sales;
ALTER TABLE sales2 RENAME TO sales;
GRANT ALL PRIVILEGES ON sales TO admin;
GRANT SELECT ON sales TO guest;
```

Limitations of Partitioned Tables

For each partition level, a partitioned table can have a maximum of 32,767 partitions.

A primary key or unique constraint on a partitioned table must contain all the partitioning columns. A unique index can omit the partitioning columns; however, it is enforced only on the parts of the partitioned table, not on the partitioned table as a whole.

Tables created with the `DISTRIBUTED REPLICATED` distribution policy cannot be partitioned.

GPORCA, the Greenplum next generation query optimizer, supports uniform multi-level partitioned tables. If GPORCA is enabled (the default) and the multi-level partitioned table is not uniform, Greenplum Database executes queries against the table with the Postgres Planner. For information about uniform multi-level partitioned tables, see *About Uniform Multi-level Partitioned Tables*.

For information about exchanging a leaf child partition with an external table, see *Exchanging a Leaf Child Partition with an External Table*.

These are limitations for partitioned tables when a leaf child partition of the table is an external table:

- Queries that run against partitioned tables that contain external table partitions are executed with the Postgres Planner.
- The external table partition is a read only external table. Commands that attempt to access or modify data in the external table partition return an error. For example:
  - `INSERT`, `DELETE`, and `UPDATE` commands that attempt to change data in the external table partition return an error.
  - `TRUNCATE` commands return an error.
  - `COPY` commands cannot copy data to a partitioned table that updates an external table partition.
  - `COPY` commands that attempt to copy from an external table partition return an error unless you specify the `IGNORE EXTERNAL PARTITIONS` clause with `COPY` command. If you specify the clause, data is not copied from external table partitions.

    To use the `COPY` command against a partitioned table with a leaf child table that is an external table, use an SQL query to copy the data. For example, if the table `my_sales` contains a with a leaf child table that is an external table, this command sends the data to `stdout`:

    ```
    COPY (SELECT * from my_sales ) TO stdout
    ```
  - `VACUUM` commands skip external table partitions.
- The following operations are supported if no data is changed on the external table partition. Otherwise, an error is returned.

  - Adding or dropping a column.
  - Changing the data type of column.
- These `ALTER PARTITION` operations are not supported if the partitioned table contains an external table partition:
  - Setting a subpartition template.
  - Altering the partition properties.
  - Creating a default partition.
  - Setting a distribution policy.
  - Setting or dropping a `NOT NULL` constraint of column.
  - Adding or dropping constraints.
  - Splitting an external partition.
- The Greenplum Database `gpbackup` utility does not back up data from a leaf child partition of a partitioned table if the leaf child partition is a readable external table.

Loading Partitioned Tables

After you create the partitioned table structure, top-level parent tables are empty. Data is routed to the bottom-level child table partitions. In a multi-level partition design, only the subpartitions at the bottom of the hierarchy can contain data.

Rows that cannot be mapped to a child table partition are rejected and the load fails. To avoid unmapped rows being rejected at load time, define your partition hierarchy with a `DEFAULT` partition. Any rows that do not match a partition's `CHECK` constraints load into the `DEFAULT` partition. See *Adding a Default Partition*.

At runtime, the query optimizer scans the entire table inheritance hierarchy and uses the `CHECK` table constraints to determine which of the child table partitions to scan to satisfy the query's conditions. The `DEFAULT` partition (if your hierarchy has one) is always scanned. `DEFAULT` partitions that contain data slow down the overall scan time.

When you use `COPY` or `INSERT` to load data into a parent table, the data is automatically rerouted to the correct partition, just like a regular table.

Best practice for loading data into partitioned tables is to create an intermediate staging table, load it, and then exchange it into your partition design. See *Exchanging a Partition*.

Verifying Your Partition Strategy

When a table is partitioned based on the query predicate, you can use `EXPLAIN` to verify that the query optimizer scans only the relevant data to examine the query plan.

For example, suppose a *sales* table is date-range partitioned by month and subpartitioned by region as shown in *Figure 20: Example Multi-level Partition Design*. For the following query:

```
EXPLAIN SELECT * FROM sales WHERE date='01-07-12' AND
region='usa';
```

The query plan for this query should show a table scan of only the following tables:

- the default partition returning 0-1 rows (if your partition design has one)
- the January 2012 partition (*sales_1_prt_1*) returning 0-1 rows
- the USA region subpartition (*sales_1_2_prt_usa*) returning *some number* of rows.

The following example shows the relevant portion of the query plan.

```
->  Seq Scan onsales_1_prt_1 sales (cost=0.00..0.00 rows=0
     width=0)
Filter: "date"=01-07-12::date AND region='USA'::text
->  Seq Scan onsales_1_2_prt_usa sales (cost=0.00..9.87
```

```
rows=20
      width=40)
```

Ensure that the query optimizer does not scan unnecessary partitions or subpartitions (for example, scans of months or regions not specified in the query predicate), and that scans of the top-level tables return 0-1 rows.

Troubleshooting Selective Partition Scanning

The following limitations can result in a query plan that shows a non-selective scan of your partition hierarchy.

- The query optimizer can selectively scan partitioned tables only when the query contains a direct and simple restriction of the table using immutable operators such as:

  =, < , <= , >, >= , and <>
- Selective scanning recognizes `STABLE` and `IMMUTABLE` functions, but does not recognize `VOLATILE` functions within a query. For example, `WHERE` clauses such as `date > CURRENT_DATE` cause the query optimizer to selectively scan partitioned tables, but `time > TIMEOFDAY` does not.

Viewing Your Partition Design

You can look up information about your partition design using the *pg_partitions* system view. For example, to see the partition design of the *sales* table:

```
SELECT partitionboundary, partitiontablename, partitionname,
partitionlevel, partitionrank
FROM pg_partitions
WHERE tablename='sales';
```

The following table and views also show information about partitioned tables.

- *pg_partition*- Tracks partitioned tables and their inheritance level relationships.
- *pg_partition_templates*- Shows the subpartitions created using a subpartition template.
- *pg_partition_columns* - Shows the partition key columns used in a partition design.

For information about Greenplum Database system catalog tables and views, see the *Greenplum Database Reference Guide*.

Maintaining Partitioned Tables

To maintain a partitioned table, use the `ALTER TABLE` command against the top-level parent table. The most common scenario is to drop old partitions and add new ones to maintain a rolling window of data in a range partition design. You can convert (*exchange*) older partitions to the append-optimized compressed storage format to save space. If you have a default partition in your partition design, you add a partition by *splitting* the default partition.

- *Adding a Partition*
- *Renaming a Partition*
- *Adding a Default Partition*
- *Dropping a Partition*
- *Truncating a Partition*
- *Exchanging a Partition*
- *Splitting a Partition*
- *Modifying a Subpartition Template*
- *Exchanging a Leaf Child Partition with an External Table*

  Important: When defining and altering partition designs, use the given partition name, not the table object name. The given partition name is the `partitionname` column value in the *pg_partitions*

system view. Although you can query and load any table (including partitioned tables) directly using SQL commands, you can only modify the structure of a partitioned table using the `ALTER TABLE...PARTITION` clauses.

Partitions are not required to have names. If a partition does not have a name, use one of the following expressions to specify a partition: `PARTITION FOR (`*`value`*`)` or `PARTITION FOR (RANK(`*`number`*`))`.

For a multi-level partitioned table, you identify a specific partition to change with `ALTER PARTITION` clauses. For each partition level in the table hierarchy that is above the target partition, specify the partition that is related to the target partition in an `ALTER PARTITION` clause. For example, if you have a partitioned table that consists of three levels, year, quarter, and region, this `ALTER TABLE` command exchanges a leaf partition `region` with the table `region_new`.

```
ALTER TABLE sales ALTER PARTITION year_1 ALTER PARTITION quarter_4 EXCHANGE
 PARTITION region WITH TABLE region_new ;
```

The two `ALTER PARTITION` clauses identify which `region` partition to exchange. Both clauses are required to identify the specific leaf partition to exchange.

Adding a Partition

You can add a partition to a partition design with the `ALTER TABLE` command. If the original partition design included subpartitions defined by a *subpartition template*, the newly added partition is subpartitioned according to that template. For example:

```
ALTER TABLE sales ADD PARTITION
            START (date '2017-02-01') INCLUSIVE
            END (date '2017-03-01') EXCLUSIVE;
```

If you did not use a subpartition template when you created the table, you define subpartitions when adding a partition:

```
ALTER TABLE sales ADD PARTITION
            START (date '2017-02-01') INCLUSIVE
            END (date '2017-03-01') EXCLUSIVE
      ( SUBPARTITION usa VALUES ('usa'),
        SUBPARTITION asia VALUES ('asia'),
        SUBPARTITION europe VALUES ('europe') );
```

When you add a subpartition to an existing partition, you can specify the partition to alter. For example:

```
ALTER TABLE sales ALTER PARTITION FOR (RANK(12))
      ADD PARTITION africa VALUES ('africa');
```

Note: You cannot add a partition to a partition design that has a default partition. You must split the default partition to add a partition. See *Splitting a Partition*.

Renaming a Partition

Partitioned tables use the following naming convention. Partitioned subtable names are subject to uniqueness requirements and length limitations.

```
<parentname>_<level>_prt_<partition_name>
```

For example:

```
sales_1_prt_jan16
```

For auto-generated range partitions, where a number is assigned when no name is given):

```
sales_1_prt_1
```

To rename a partitioned child table, rename the top-level parent table. The *<parentname>* changes in the table names of all associated child table partitions. For example, the following command:

```
ALTER TABLE sales RENAME TO globalsales;
```

Changes the associated table names:

```
globalsales_1_prt_1
```

You can change the name of a partition to make it easier to identify. For example:

```
ALTER TABLE sales RENAME PARTITION FOR ('2016-01-01') TO jan16;
```

Changes the associated table name as follows:

```
sales_1_prt_jan16
```

When altering partitioned tables with the `ALTER TABLE` command, always refer to the tables by their partition name (*jan16*) and not their full table name (*sales_1_prt_jan16*).

> **Note:** The table name cannot be a partition name in an `ALTER TABLE` statement. For example, `ALTER TABLE sales...` is correct, `ALTER TABLE sales_1_part_jan16...` is not allowed.

Adding a Default Partition

You can add a default partition to a partition design with the `ALTER TABLE` command.

```
ALTER TABLE sales ADD DEFAULT PARTITION other;
```

If your partition design is multi-level, each level in the hierarchy must have a default partition. For example:

```
ALTER TABLE sales ALTER PARTITION FOR (RANK(1)) ADD DEFAULT
PARTITION other;

ALTER TABLE sales ALTER PARTITION FOR (RANK(2)) ADD DEFAULT
PARTITION other;

ALTER TABLE sales ALTER PARTITION FOR (RANK(3)) ADD DEFAULT
PARTITION other;
```

If incoming data does not match a partition's `CHECK` constraint and there is no default partition, the data is rejected. Default partitions ensure that incoming data that does not match a partition is inserted into the default partition.

Dropping a Partition

You can drop a partition from your partition design using the `ALTER TABLE` command. When you drop a partition that has subpartitions, the subpartitions (and all data in them) are automatically dropped as well. For range partitions, it is common to drop the older partitions from the range as old data is rolled out of the data warehouse. For example:

```
ALTER TABLE sales DROP PARTITION FOR (RANK(1));
```

Truncating a Partition

You can truncate a partition using the `ALTER TABLE` command. When you truncate a partition that has subpartitions, the subpartitions are automatically truncated as well.

```
ALTER TABLE sales TRUNCATE PARTITION FOR (RANK(1));
```

Exchanging a Partition

You can exchange a partition using the `ALTER TABLE` command. Exchanging a partition swaps one table in place of an existing partition. You can exchange partitions only at the lowest level of your partition hierarchy (only partitions that contain data can be exchanged).

You cannot exchange a partition with a replicated table. Exchanging a partition with a partitioned table or a child partition of a partitioned table is not supported.

Partition exchange can be useful for data loading. For example, load a staging table and swap the loaded table into your partition design. You can use partition exchange to change the storage type of older partitions to append-optimized tables. For example:

```
CREATE TABLE jan12 (LIKE sales) WITH (appendoptimized=true);
INSERT INTO jan12 SELECT * FROM sales_1_prt_1 ;
ALTER TABLE sales EXCHANGE PARTITION FOR (DATE '2012-01-01')
WITH TABLE jan12;
```

> **Note:** This example refers to the single-level definition of the table `sales`, before partitions were added and altered in the previous examples.
>
> **Warning:** If you specify the `WITHOUT VALIDATION` clause, you must ensure that the data in table that you are exchanging for an existing partition is valid against the constraints on the partition. Otherwise, queries against the partitioned table might return incorrect results.

The Greenplum Database server configuration parameter `gp_enable_exchange_default_partition` controls availability of the `EXCHANGE DEFAULT PARTITION` clause. The default value for the parameter is `off`, the clause is not available and Greenplum Database returns an error if the clause is specified in an `ALTER TABLE` command.

For information about the parameter, see "Server Configuration Parameters" in the *Greenplum Database Reference Guide*.

> **Warning:** Before you exchange the default partition, you must ensure the data in the table to be exchanged, the new default partition, is valid for the default partition. For example, the data in the new default partition must not contain data that would be valid in other leaf child partitions of the partitioned table. Otherwise, queries against the partitioned table with the exchanged default partition that are executed by GPORCA might return incorrect results.

Splitting a Partition

Splitting a partition divides a partition into two partitions. You can split a partition using the `ALTER TABLE` command. You can split partitions only at the lowest level of your partition hierarchy (partitions that contain data). For a multi-level partition, only range partitions can be split, not list partitions. The split value you specify goes into the *latter* partition.

For example, to split a monthly partition into two with the first partition containing dates January 1-15 and the second partition containing dates January 16-31:

```
ALTER TABLE sales SPLIT PARTITION FOR ('2017-01-01')
AT ('2017-01-16')
INTO (PARTITION jan171to15, PARTITION jan1716to31);
```

If your partition design has a default partition, you must split the default partition to add a partition.

When using the `INTO` clause, specify the current default partition as the second partition name. For example, to split a default range partition to add a new monthly partition for January 2017:

```
ALTER TABLE sales SPLIT DEFAULT PARTITION
START ('2017-01-01') INCLUSIVE
END ('2017-02-01') EXCLUSIVE
INTO (PARTITION jan17, default partition);
```

Modifying a Subpartition Template

Use `ALTER TABLE` SET SUBPARTITION TEMPLATE to modify the subpartition template of a partitioned table. Partitions added after you set a new subpartition template have the new partition design. Existing partitions are not modified.

The following example alters the subpartition template of this partitioned table:

```
CREATE TABLE sales (trans_id int, date date, amount decimal(9,2), region
 text)
  DISTRIBUTED BY (trans_id)
  PARTITION BY RANGE (date)
  SUBPARTITION BY LIST (region)
  SUBPARTITION TEMPLATE
    ( SUBPARTITION usa VALUES ('usa'),
      SUBPARTITION asia VALUES ('asia'),
      SUBPARTITION europe VALUES ('europe'),
      DEFAULT SUBPARTITION other_regions )
  ( START (date '2014-01-01') INCLUSIVE
    END (date '2014-04-01') EXCLUSIVE
    EVERY (INTERVAL '1 month') );
```

This `ALTER TABLE` command, modifies the subpartition template.

```
ALTER TABLE sales SET SUBPARTITION TEMPLATE
( SUBPARTITION usa VALUES ('usa'),
  SUBPARTITION asia VALUES ('asia'),
  SUBPARTITION europe VALUES ('europe'),
  SUBPARTITION africa VALUES ('africa'),
  DEFAULT SUBPARTITION regions );
```

When you add a date-range partition of the table sales, it includes the new regional list subpartition for Africa. For example, the following command creates the subpartitions `usa`, `asia`, `europe`, `africa`, and a default partition named `other`:

```
ALTER TABLE sales ADD PARTITION "4"
  START ('2014-04-01') INCLUSIVE
  END ('2014-05-01') EXCLUSIVE ;
```

To view the tables created for the partitioned table `sales`, you can use the command `\dt sales*` from the psql command line.

To remove a subpartition template, use `SET SUBPARTITION TEMPLATE` with empty parentheses. For example, to clear the sales table subpartition template:

```
ALTER TABLE sales SET SUBPARTITION TEMPLATE ();
```

Exchanging a Leaf Child Partition with an External Table

You can exchange a leaf child partition of a partitioned table with a readable external table. The external table data can reside on a host file system, an NFS mount, or a Hadoop file system (HDFS).

For example, if you have a partitioned table that is created with monthly partitions and most of the queries against the table only access the newer data, you can copy the older, less accessed data to external tables and exchange older partitions with the external tables. For queries that only access the newer data, you could create queries that use partition elimination to prevent scanning the older, unneeded partitions.

Exchanging a leaf child partition with an external table is not supported if the partitioned table contains a column with a check constraint or a `NOT NULL` constraint.

For information about exchanging and altering a leaf child partition, see the `ALTER TABLE` command in the *Greenplum Database Command Reference*.

For information about limitations of partitioned tables that contain a external table partition, see *Limitations of Partitioned Tables*.

Example Exchanging a Partition with an External Table

This is a simple example that exchanges a leaf child partition of this partitioned table for an external table. The partitioned table contains data for the years 2010 through 2013.

```
CREATE TABLE sales (id int, year int, qtr int, day int, region text)
  DISTRIBUTED BY (id)
  PARTITION BY RANGE (year)
  ( PARTITION yr START (2010) END (2014) EVERY (1) ) ;
```

There are four leaf child partitions for the partitioned table. Each leaf child partition contains the data for a single year. The leaf child partition table `sales_1_prt_yr_1` contains the data for the year 2010. These steps exchange the table `sales_1_prt_yr_1` with an external table the uses the `gpfdist` protocol:

1. Ensure that the external table protocol is enabled for the Greenplum Database system.

   This example uses the `gpfdist` protocol. This command starts the `gpfdist` protocol.

   ```
   $ gpfdist
   ```

2. Create a writable external table.

   This `CREATE WRITABLE EXTERNAL TABLE` command creates a writable external table with the same columns as the partitioned table.

   ```
   CREATE WRITABLE EXTERNAL TABLE my_sales_ext ( LIKE sales_1_prt_yr_1 )
     LOCATION ( 'gpfdist://gpdb_test/sales_2010' )
     FORMAT 'csv'
     DISTRIBUTED BY (id) ;
   ```

3. Create a readable external table that reads the data from that destination of the writable external table created in the previous step.

   This `CREATE EXTERNAL TABLE` create a readable external that uses the same external data as the writable external data.

   ```
   CREATE EXTERNAL TABLE sales_2010_ext ( LIKE sales_1_prt_yr_1)
     LOCATION ( 'gpfdist://gpdb_test/sales_2010' )
     FORMAT 'csv' ;
   ```

4. Copy the data from the leaf child partition into the writable external table.

This INSERT command copies the data from the child leaf partition table of the partitioned table into the external table.

```
INSERT INTO my_sales_ext SELECT * FROM sales_1_prt_yr_1 ;
```

5. Exchange the existing leaf child partition with the external table.

   This ALTER TABLE command specifies the EXCHANGE PARTITION clause to switch the readable external table and the leaf child partition.

   ```
   ALTER TABLE sales ALTER PARTITION yr_1
      EXCHANGE PARTITION yr_1
      WITH TABLE sales_2010_ext WITHOUT VALIDATION;
   ```

   The external table becomes the leaf child partition with the table name sales_1_prt_yr_1 and the old leaf child partition becomes the table sales_2010_ext.

   Warning: In order to ensure queries against the partitioned table return the correct results, the external table data must be valid against the CHECK constraints on the leaf child partition. In this case, the data was taken from the child leaf partition table on which the CHECK constraints were defined.

6. Drop the table that was rolled out of the partitioned table.

   ```
   DROP TABLE sales_2010_ext ;
   ```

You can rename the name of the leaf child partition to indicate that sales_1_prt_yr_1 is an external table.

This example command changes the partitionname to yr_1_ext and the name of the child leaf partition table to sales_1_prt_yr_1_ext.

```
ALTER TABLE sales RENAME PARTITION yr_1 TO  yr_1_ext ;
```

Creating and Using Sequences

A Greenplum Database sequence object is a special single row table that functions as a number generator. You can use a sequence to generate unique integer identifiers for a row that you add to a table. Declaring a column of type SERIAL implicitly creates a sequence counter for use in that table column.

Greenplum Database provides commands to create, alter, and drop a sequence. Greenplum Database also provides built-in functions to return the next value in the sequence (nextval()) or to set the sequence to a specific start value (setval()).

Note: The PostgreSQL currval() and lastval() sequence functions are not supported in Greenplum Database.

Attributes of a sequence object include the name of the sequence, its increment value, and the last, minimum, and maximum values of the sequence counter. Sequences also have a special boolean attribute named is_called that governs the auto-increment behavior of a nextval() operation on the sequence counter. When a sequence's is_called attribute is true, nextval() increments the sequence counter before returning the value. When the is_called attribute value of a sequence is false, nextval() does not increment the counter before returning the value.

Creating a Sequence

The *CREATE SEQUENCE* command creates and initializes a sequence with the given sequence name and optional start value. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema. For example:

```
CREATE SEQUENCE myserial START 101;
```

When you create a new sequence, Greenplum Database sets the sequence `is_called` attribute to `false`. Invoking `nextval()` on a newly-created sequence does not increment the sequence counter, but returns the sequence start value and sets `is_called` to `true`.

Using a Sequence

After you create a sequence with the `CREATE SEQUENCE` command, you can examine the sequence and use the sequence built-in functions.

Examining Sequence Attributes

To examine the current attributes of a sequence, query the sequence directly. For example, to examine a sequence named `myserial`:

```
SELECT * FROM myserial;
```

Returning the Next Sequence Counter Value

You can invoke the `nextval()` built-in function to return and use the next value in a sequence. The following command inserts the next value of the sequence named `myserial` into the first column of a table named `vendors`:

```
INSERT INTO vendors VALUES (nextval('myserial'), 'acme');
```

`nextval()` uses the sequence's `is_called` attribute value to determine whether or not to increment the sequence counter before returning the value. `nextval()` advances the counter when `is_called` is `true`. `nextval()` sets the sequence `is_called` attribute to `true` before returning.

A `nextval()` operation is never rolled back. A fetched value is considered used, even if the transaction that performed the `nextval()` fails. This means that failed transactions can leave unused holes in the sequence of assigned values.

> **Note:** You cannot use the `nextval()` function in `UPDATE` or `DELETE` statements if mirroring is enabled in Greenplum Database.

Setting the Sequence Counter Value

You can use the Greenplum Database `setval()` built-in function to set the counter value for a sequence. For example, the following command sets the counter value of the sequence named `myserial` to `201`:

```
SELECT setval('myserial', 201);
```

`setval()` has two function signatures: `setval(sequence, start_val)` and `setval(sequence, start_val, is_called)`. The default behaviour of `setval(sequence, start_val)` sets the sequence `is_called` attribute value to `true`.

If you do not want the sequence counter advanced on the next `nextval()` call, use the `setval(sequence, start_val, is_called)` function signature, passing a `false` argument:

```
SELECT setval('myserial', 201, false);
```

`setval()` operations are never rolled back.

Altering a Sequence

The *`ALTER SEQUENCE`* command changes the attributes of an existing sequence. You can alter the sequence start, minimum, maximum, and increment values. You can also restart the sequence at the start value or at a specified value.

Any parameters not set in the `ALTER SEQUENCE` command retain their prior settings.

`ALTER SEQUENCE` *`sequence`* `START WITH` *`start_value`* sets the sequence's `start_value` attribute to the new starting value. It has no effect on the `last_value` attribute or the value returned by the `nextval(sequence)` function.

`ALTER SEQUENCE` *`sequence`* `RESTART` resets the sequence's `last_value` attribute to the current value of the `start_value` attribute and the `is_called` attribute to `false`. The next call to the `nextval(sequence)` function returns *start_value*.

`ALTER SEQUENCE` *`sequence`* `RESTART WITH` *`restart_value`* sets the sequence's *last_value* attribute to the new value and the `is_called` attribute to `false`. The next call to the `nextval(sequence)` returns *restart_value*. This is the equivalent of calling `setval(sequence, restart_value, false)`.

The following command restarts the sequence named `myserial` at value `105`:

```
ALTER SEQUENCE myserial RESTART WITH 105;
```

Dropping a Sequence

The *`DROP SEQUENCE`* command removes a sequence. For example, the following command removes the sequence named `myserial`:

```
DROP SEQUENCE myserial;
```

Specifying a Sequence as the Default Value for a Column

You can reference a sequence directly in the *`CREATE TABLE`* command in addition to using the `SERIAL` or `BIGSERIAL` types. For example:

```
CREATE TABLE tablename ( id INT4 DEFAULT nextval('myserial'), name text );
```

You can also alter a table column to set its default value to a sequence counter:

```
ALTER TABLE tablename ALTER COLUMN id SET DEFAULT nextval('myserial');
```

Sequence Wraparound

By default, a sequence does not wrap around. That is, when a sequence reaches the max value (`+32767` for `SMALLSERIAL`, `+2147483647` for `SERIAL`, `+9223372036854775807` for `BIGSERIAL`), every subsequent `nextval()` call produces an error. You can alter a sequence to make it cycle around and start at `1` again:

```
ALTER SEQUENCE myserial CYCLE;
```

You can also specify the wraparound behaviour when you create the sequence:

```
CREATE SEQUENCE myserial CYCLE;
```

Using Indexes in Greenplum Database

In most traditional databases, indexes can greatly improve data access times. However, in a distributed database such as Greenplum, indexes should be used more sparingly. Greenplum Database performs very fast sequential scans; indexes use a random seek pattern to locate records on disk. Greenplum data is distributed across the segments, so each segment scans a smaller portion of the overall data to get the result. With table partitioning, the total data to scan may be even smaller. Because business intelligence (BI) query workloads generally return very large data sets, using indexes is not efficient.

First try your query workload without adding indexes. Indexes are more likely to improve performance for OLTP workloads, where the query is returning a single record or a small subset of data. Indexes can also improve performance on compressed append-optimized tables for queries that return a targeted set of rows, as the optimizer can use an index access method rather than a full table scan when appropriate. For compressed data, an index access method means only the necessary rows are uncompressed.

Greenplum Database automatically creates `PRIMARY KEY` constraints for tables with primary keys. To create an index on a partitioned table, create an index on the partitioned table that you created. The index is propagated to all the child tables created by Greenplum Database. Creating an index on a table that is created by Greenplum Database for use by a partitioned table is not supported.

Note that a `UNIQUE CONSTRAINT` (such as a `PRIMARY KEY CONSTRAINT`) implicitly creates a `UNIQUE INDEX` that must include all the columns of the distribution key and any partitioning key. The `UNIQUE CONSTRAINT` is enforced across the entire table, including all table partitions (if any).

Indexes add some database overhead — they use storage space and must be maintained when the table is updated. Ensure that the query workload uses the indexes that you create, and check that the indexes you add improve query performance (as compared to a sequential scan of the table). To determine whether indexes are being used, examine the query `EXPLAIN` plans. See *Query Profiling*.

Consider the following points when you create indexes.

- **Your Query Workload.** Indexes improve performance for workloads where queries return a single record or a very small data set, such as OLTP workloads.
- Compressed Tables. Indexes can improve performance on compressed append-optimized tables for queries that return a targeted set of rows. For compressed data, an index access method means only the necessary rows are uncompressed.
- **Avoid indexes on frequently updated columns.** Creating an index on a column that is frequently updated increases the number of writes required when the column is updated.
- **Create selective B-tree indexes.** Index selectivity is a ratio of the number of distinct values a column has divided by the number of rows in a table. For example, if a table has 1000 rows and a column has 800 distinct values, the selectivity of the index is 0.8, which is considered good. Unique indexes always have a selectivity ratio of 1.0, which is the best possible. Greenplum Database allows unique indexes only on distribution key columns.
- **Use Bitmap indexes for low selectivity columns.** The Greenplum Database Bitmap index type is not available in regular PostgreSQL. See *About Bitmap Indexes*.
- **Index columns used in joins.** An index on a column used for frequent joins (such as a foreign key column) can improve join performance by enabling more join methods for the query optimizer to use.
- **Index columns frequently used in predicates.** Columns that are frequently referenced in `WHERE` clauses are good candidates for indexes.
- **Avoid overlapping indexes.** Indexes that have the same leading column are redundant.
- **Drop indexes for bulk loads.** For mass loads of data into a table, consider dropping the indexes and re-creating them after the load completes. This is often faster than updating the indexes.

- **Consider a clustered index.** Clustering an index means that the records are physically ordered on disk according to the index. If the records you need are distributed randomly on disk, the database has to seek across the disk to fetch the records requested. If the records are stored close together, the fetching operation is more efficient. For example, a clustered index on a date column where the data is ordered sequentially by date. A query against a specific date range results in an ordered fetch from the disk, which leverages fast sequential access.

To cluster an index in Greenplum Database

Using the `CLUSTER` command to physically reorder a table based on an index can take a long time with very large tables. To achieve the same results much faster, you can manually reorder the data on disk by creating an intermediate table and loading the data in the desired order. For example:

```
CREATE TABLE new_table (LIKE old_table)
       AS SELECT * FROM old_table ORDER BY myixcolumn;
DROP old_table;
ALTER TABLE new_table RENAME TO old_table;
CREATE INDEX myixcolumn_ix ON old_table;
VACUUM ANALYZE old_table;
```

Index Types

Greenplum Database supports the Postgres index types B-tree, GiST, SP-GiST, and GIN. Hash indexes are not supported. Each index type uses a different algorithm that is best suited to different types of queries. B-tree indexes fit the most common situations and are the default index type. See *Index Types* in the PostgreSQL documentation for a description of these types.

> **Note:** Greenplum Database allows unique indexes only if the columns of the index key are the same as (or a superset of) the Greenplum distribution key. Unique indexes are not supported on append-optimized tables. On partitioned tables, a unique index cannot be enforced across all child table partitions of a partitioned table. A unique index is supported only within a partition.

About Bitmap Indexes

Greenplum Database provides the Bitmap index type. Bitmap indexes are best suited to data warehousing applications and decision support systems with large amounts of data, many ad hoc queries, and few data modification (DML) transactions.

An index provides pointers to the rows in a table that contain a given key value. A regular index stores a list of tuple IDs for each key corresponding to the rows with that key value. Bitmap indexes store a bitmap for each key value. Regular indexes can be several times larger than the data in the table, but bitmap indexes provide the same functionality as a regular index and use a fraction of the size of the indexed data.

Each bit in the bitmap corresponds to a possible tuple ID. If the bit is set, the row with the corresponding tuple ID contains the key value. A mapping function converts the bit position to a tuple ID. Bitmaps are compressed for storage. If the number of distinct key values is small, bitmap indexes are much smaller, compress better, and save considerable space compared with a regular index. The size of a bitmap index is proportional to the number of rows in the table times the number of distinct values in the indexed column.

Bitmap indexes are most effective for queries that contain multiple conditions in the `WHERE` clause. Rows that satisfy some, but not all, conditions are filtered out before the table is accessed. This improves response time, often dramatically.

When to Use Bitmap Indexes

Bitmap indexes are best suited to data warehousing applications where users query the data rather than update it. Bitmap indexes perform best for columns that have between 100 and 100,000 distinct values and when the indexed column is often queried in conjunction with other indexed columns. Columns with fewer than 100 distinct values, such as a gender column with two distinct values (male and female), usually

do not benefit much from any type of index. On a column with more than 100,000 distinct values, the performance and space efficiency of a bitmap index decline.

Bitmap indexes can improve query performance for ad hoc queries. `AND` and `OR` conditions in the `WHERE` clause of a query can be resolved quickly by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to tuple ids. If the resulting number of rows is small, the query can be answered quickly without resorting to a full table scan.

When Not to Use Bitmap Indexes

Do not use bitmap indexes for unique columns or columns with high cardinality data, such as customer names or phone numbers. The performance gains and disk space advantages of bitmap indexes start to diminish on columns with 100,000 or more unique values, regardless of the number of rows in the table.

Bitmap indexes are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data.

Use bitmap indexes sparingly. Test and compare query performance with and without an index. Add an index only if query performance improves with indexed columns.

Creating an Index

The `CREATE INDEX` command defines an index on a table. A B-tree index is the default index type. For example, to create a B-tree index on the column *gender* in the table *employee*:

```
CREATE INDEX gender_idx ON employee (gender);
```

To create a bitmap index on the column *title* in the table *films*:

```
CREATE INDEX title_bmp_idx ON films USING bitmap (title);
```

Indexes on Expressions

An index column need not be just a column of the underlying table, but can be a function or scalar expression computed from one or more columns of the table. This feature is useful to obtain fast access to tables based on the results of computations.

Index expressions are relatively expensive to maintain, because the derived expressions must be computed for each row upon insertion and whenever it is updated. However, the index expressions are not recomputed during an indexed search, since they are already stored in the index. In both of the following examples, the system sees the query as just `WHERE indexedcolumn = 'constant'` and so the speed of the search is equivalent to any other simple index query. Thus, indexes on expressions are useful when retrieval speed is more important than insertion and update speed.

The first example is a common way to do case-insensitive comparisons with the `lower` function:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

This query can use an index if one has been defined on the result of the `lower(col1)` function:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

This example assumes the following type of query is performed often.

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

The query might benefit from the following index.

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

The syntax of the `CREATE INDEX` command normally requires writing parentheses around index expressions, as shown in the second example. The parentheses can be omitted when the expression is just a function call, as in the first example.

Examining Index Usage

Greenplum Database indexes do not require maintenance and tuning. You can check which indexes are used by the real-life query workload. Use the `EXPLAIN` command to examine index usage for a query.

The query plan shows the steps or *plan nodes* that the database will take to answer a query and time estimates for each plan node. To examine the use of indexes, look for the following query plan node types in your `EXPLAIN` output:

- **Index Scan** - A scan of an index.
- **Bitmap Heap Scan** - Retrieves all
- from the bitmap generated by BitmapAnd, BitmapOr, or BitmapIndexScan and accesses the heap to retrieve the relevant rows.
- **Bitmap Index Scan** - Compute a bitmap by OR-ing all bitmaps that satisfy the query predicates from the underlying index.
- **BitmapAnd** or **BitmapOr** - Takes the bitmaps generated from multiple BitmapIndexScan nodes, ANDs or ORs them together, and generates a new bitmap as its output.

You have to experiment to determine the indexes to create. Consider the following points.

- Run `ANALYZE` after you create or update an index. `ANALYZE` collects table statistics. The query optimizer uses table statistics to estimate the number of rows returned by a query and to assign realistic costs to each possible query plan.
- Use real data for experimentation. Using test data for setting up indexes tells you what indexes you need for the test data, but that is all.
- Do not use very small test data sets as the results can be unrealistic or skewed.
- Be careful when developing test data. Values that are similar, completely random, or inserted in sorted order will skew the statistics away from the distribution that real data would have.
- You can force the use of indexes for testing purposes by using run-time parameters to turn off specific plan types. For example, turn off sequential scans (`enable_seqscan`) and nested-loop joins (`enable_nestloop`), the most basic plans, to force the system to use a different plan. Time your query with and without indexes and use the `EXPLAIN ANALYZE` command to compare the results.

Managing Indexes

Use the `REINDEX` command to rebuild a poorly-performing index. `REINDEX` rebuilds an index using the data stored in the index's table, replacing the old copy of the index.

To rebuild all indexes on a table

```
REINDEX my_table;
```

To rebuild a particular index

```
REINDEX my_index;
```

Dropping an Index

The `DROP INDEX` command removes an index. For example:

```
DROP INDEX title_idx;
```

When loading data, it can be faster to drop all indexes, load, then recreate the indexes.

Creating and Managing Views

Views enable you to save frequently used or complex queries, then access them in a `SELECT` statement as if they were a table. A view is not physically materialized on disk: the query runs as a subquery when you access the view.

These topics describe various aspects of creating and managing views:

- *Best Practices when Creating Views* outlines best practices when creating views.
- *Working with View Dependencies* contains examples of listing view information and determining what views depend on a certain object.
- *About View Storage in Greenplum Database* describes the mechanics behind view dependencies.

Creating Views

The `CREATE VIEW` command defines a view of a query. For example:

```
CREATE VIEW comedies AS SELECT * FROM films WHERE kind = 'comedy';
```

Views ignore `ORDER BY` and `SORT` operations stored in the view.

Dropping Views

The `DROP VIEW` command removes a view. For example:

```
DROP VIEW topten;
```

The `DROP VIEW...CASCADE` command also removes all dependent objects. As an example, if another view depends on the view which is about to be dropped, the other view will be dropped as well. Without the `CASCADE` option, the `DROP VIEW` command will fail.

Best Practices when Creating Views

When defining and using a view, remember that a view is just an SQL statement and is replaced by its definition when the query is executed.

These are some common uses of views.

- They allow you to have a recurring SQL query or expression in one place for easy reuse.
- They can be used as an interface to abstract from the actual table definitions, so that you can reorganize the tables without having to modify the interface.

If a subquery is associated with a single query, consider using the `WITH` clause of the `SELECT` command instead of creating a seldom-used view.

In general, these uses do not require nesting views, that is, defining views based on other views.

These are two patterns of creating views that tend to be problematic because the view's SQL is used during query execution.

- Defining many layers of views so that your final queries look deceptively simple.

  Problems arise when you try to enhance or troubleshoot queries that use the views, for example by examining the execution plan. The query's execution plan tends to be complicated and it is difficult to understand and how to improve it.
- Defining a denormalized "world" view. A view that joins a large number of database tables that is used for a wide variety of queries.

Performance issues can occur for some queries that use the view for some `WHERE` conditions while other `WHERE` conditions work well.

Working with View Dependencies

If there are view dependencies on a table you must use the `CASCADE` keyword to drop it. Also, you cannot alter the table if there are view dependencies on it. This example shows a view dependency on a table.

```
CREATE TABLE t (id integer PRIMARY KEY);
CREATE VIEW v AS SELECT * FROM t;

DROP TABLE t;
ERROR:  cannot drop table t because other objects depend on it
DETAIL:  view v depends on table t
HINT:  Use DROP ... CASCADE to drop the dependent objects too.

ALTER TABLE t DROP id;
ERROR:  cannot drop column id of table t because other objects depend on it
DETAIL:  view v depends on column id of table t
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

As the previous example shows, altering a table can be quite a challenge if there is a deep hierarchy of views, because you have to create the views in the correct order. You cannot create a view unless all the objects it requires are present.

You can use view dependency information when you want to alter a table that is referenced by a view. For example, you might want to change a table's column data type from `integer` to `bigint` because you realize you need to store larger numbers. However, you cannot do that if there are views that use the column. You first have to drop those views, then change the column and then run all the `CREATE VIEW` statements to create the views again.

Finding View Dependencies

The following example queries list view information on dependencies on tables and columns.

- *Finding Direct View Dependencies on a Table*
- *Finding Direct Dependencies on a Table Column*
- *Listing View Schemas*
- *Listing View Definitions*
- *Listing Nested Views*

The example output is based on the *Example Data* at the end of this topic.

Also, you can use the first example query *Finding Direct View Dependencies on a Table* to find dependencies on user-defined functions (or procedures). The query uses the catalog table `pg_class` that contains information about tables and views. For functions, you can use the catalog table *`pg_proc`* to get information about functions.

For detailed information about the system catalog tables that store view information, see *About View Storage in Greenplum Database*.

Finding Direct View Dependencies on a Table

To find out which views directly depend on table `t1`, create a query that performs a join among the catalog tables that contain the dependency information, and qualify the query to return only view dependencies.

```
SELECT v.oid::regclass AS view,
  d.refobjid::regclass AS ref_object    -- name of table
  -- d.refobjid::regproc AS ref_object  -- name of function
FROM pg_depend AS d      -- objects that depend on a table
```

```
  JOIN pg_rewrite AS r  -- rules depending on a table
     ON r.oid = d.objid
  JOIN pg_class AS v    -- views for the rules
     ON v.oid = r.ev_class
WHERE v.relkind = 'v'         -- filter views only
  -- dependency must be a rule depending on a relation
  AND d.classid = 'pg_rewrite'::regclass
  AND d.deptype = 'n'         -- normal dependency
  -- qualify object
  AND d.refclassid = 'pg_class'::regclass   -- dependent table
  AND d.refobjid = 't1'::regclass
  -- AND d.refclassid = 'pg_proc'::regclass -- dependent function
  -- AND d.refobjid = 'f'::regproc
;
    view    | ref_object
------------+------------
 v1         | t1
 v2         | t1
 v2         | t1
 v3         | t1
 mytest.vt1 | t1
 mytest.v2a | t1
 mytest.v2a | t1
(7 rows)
```

The query performs casts to the `regclass` object identifier type. For information about object identifier types, see the PostgeSQL documentation on *Object Identifier Types*.

In some cases, the views are listed multiple times because the view references multiple table columns. You can remove those duplicates using `DISTINCT`.

You can alter the query to find views with direct dependencies on the function `f`.

- In the `SELECT` clause replace the name of the table `d.refobjid::regclass as ref_object` with the name of the function `d.refobjid::regproc as ref_object`
- In the `WHERE` clause replace the catalog of the referenced object from `d.refclassid = 'pg_class'::regclass` for tables, to `d.refclassid = 'pg_proc'::regclass` for procedures (functions). Also change the object name from `d.refobjid = 't1'::regclass` to `d.refobjid = 'f'::regproc`
- In the `WHERE` clause, replace the name of the table `refobjid = 't1'::regclass` with the name of the function `refobjid = 'f'::regproc`.

In the example query, the changes have been commented out (prefixed with `--`). You can comment out the lines for the table and enable the lines for the function.

Finding Direct Dependencies on a Table Column

You can modify the previous query to find those views that depend on a certain table column, which can be useful if you are planning to drop a column (adding a column to the base table is never a problem). The query uses the table column information in the *`pg_attribute`* catalog table.

This query finds the views that depend on the column `id` of table `t1`:

```
SELECT v.oid::regclass AS view,
  d.refobjid::regclass AS ref_object, -- name of table
  a.attname AS col_name               -- column name
FROM pg_attribute AS a   -- columns for a table
  JOIN pg_depend AS d    -- objects that depend on a column
    ON d.refobjsubid = a.attnum AND d.refobjid = a.attrelid
  JOIN pg_rewrite AS r   -- rules depending on the column
    ON r.oid = d.objid
  JOIN pg_class AS v     -- views for the rules
    ON v.oid = r.ev_class
```

```
WHERE v.relkind = 'v'    -- filter views only
  -- dependency must be a rule depending on a relation
  AND d.classid = 'pg_rewrite'::regclass
  AND d.refclassid = 'pg_class'::regclass
  AND d.deptype = 'n'    -- normal dependency
  AND a.attrelid = 't1'::regclass
  AND a.attname = 'id'
;
    view    | ref_object | col_name
------------+------------+----------
 v1         | t1         | id
 v2         | t1         | id
 mytest.vt1 | t1         | id
 mytest.v2a | t1         | id
(4 rows)
```

Listing View Schemas

If you have created views in multiple schemas, you can also list views, each view's schema, and the table referenced by the view. The query retrieves the schema from the catalog table `pg_namespace` and excludes the system schemas `pg_catalog`, `information_schema`, and `gp_toolkit`. Also, the query does not list a view if the view refers to itself.

```
SELECT v.oid::regclass AS view,
  ns.nspname AS schema,       -- view schema,
  d.refobjid::regclass AS ref_object -- name of table
FROM pg_depend AS d            -- objects that depend on a table
  JOIN pg_rewrite AS r        -- rules depending on a table
    ON r.oid = d.objid
  JOIN pg_class AS v          -- views for the rules
    ON v.oid = r.ev_class
  JOIN pg_namespace AS ns     -- schema information
    ON ns.oid = v.relnamespace
WHERE v.relkind = 'v'          -- filter views only
  -- dependency must be a rule depending on a relation
  AND d.classid = 'pg_rewrite'::regclass
  AND d.refclassid = 'pg_class'::regclass  -- referenced objects in pg_class
 (tables and views)
  AND d.deptype = 'n'         -- normal dependency
  -- qualify object
  AND ns.nspname NOT IN ('pg_catalog', 'information_schema', 'gp_toolkit')
 -- system schemas
  AND NOT (v.oid = d.refobjid) -- not self-referencing dependency
;
    view    | schema | ref_object
------------+--------+------------
 v1         | public | t1
 v2         | public | t1
 v2         | public | t1
 v2         | public | v1
 v3         | public | t1
 vm1        | public | mytest.tm1
 mytest.vm1 | mytest | t1
 vm2        | public | mytest.tm1
 mytest.v2a | mytest | t1
 mytest.v2a | mytest | t1
 mytest.v2a | mytest | v1
(11 rows)
```

Listing View Definitions

This query lists the views that depend on `t1`, the column referenced, and the view definition. The `CREATE VIEW` command is created by adding the appropriate text to the view definition.

```
SELECT v.relname AS view,
  d.refobjid::regclass as ref_object,
  d.refobjsubid as ref_col,
  'CREATE VIEW ' || v.relname || ' AS ' || pg_get_viewdef(v.oid) AS view_def
FROM pg_depend AS d
  JOIN pg_rewrite AS r
    ON r.oid = d.objid
  JOIN pg_class AS v
    ON v.oid = r.ev_class
WHERE NOT (v.oid = d.refobjid)
  AND d.refobjid = 't1'::regclass
  ORDER BY d.refobjsubid
;
 view | ref_object | ref_col |                  view_def
------+------------+---------+-------------------------------------------
 v1   | t1         |       1 | CREATE VIEW v1 AS  SELECT max(t1.id) AS id+
      |            |         |    FROM t1;
 v2a  | t1         |       1 | CREATE VIEW v2a AS  SELECT t1.val         +
      |            |         |    FROM (t1                               +
      |            |         |      JOIN v1 USING (id));
 vt1  | t1         |       1 | CREATE VIEW vt1 AS  SELECT t1.id          +
      |            |         |    FROM t1                                +
      |            |         |   WHERE (t1.id < 3);
 v2   | t1         |       1 | CREATE VIEW v2 AS  SELECT t1.val          +
      |            |         |    FROM (t1                               +
      |            |         |      JOIN v1 USING (id));
 v2a  | t1         |       2 | CREATE VIEW v2a AS  SELECT t1.val         +
      |            |         |    FROM (t1                               +
      |            |         |      JOIN v1 USING (id));
 v3   | t1         |       2 | CREATE VIEW v3 AS  SELECT (t1.val || f()) +
      |            |         |    FROM t1;
 v2   | t1         |       2 | CREATE VIEW v2 AS  SELECT t1.val          +
      |            |         |    FROM (t1                               +
      |            |         |      JOIN v1 USING (id));
(7 rows)
```

Listing Nested Views

This CTE query lists information about views that reference another view.

The `WITH` clause in this CTE query selects all the views in the user schemas. The main `SELECT` statement finds all views that reference another view.

```
WITH views AS ( SELECT v.relname AS view,
  d.refobjid AS ref_object,
  v.oid AS view_oid,
  ns.nspname AS namespace
FROM pg_depend AS d
  JOIN pg_rewrite AS r
    ON r.oid = d.objid
  JOIN pg_class AS v
    ON v.oid = r.ev_class
  JOIN pg_namespace AS ns
    ON ns.oid = v.relnamespace
WHERE v.relkind = 'v'
  AND ns.nspname NOT IN ('pg_catalog', 'information_schema', 'gp_toolkit')
 -- exclude system schemas
  AND d.deptype = 'n'    -- normal dependency
```

```
  AND NOT (v.oid = d.refobjid) -- not a self-referencing dependency
 )
SELECT views.view, views.namespace AS schema,
  views.ref_object::regclass AS ref_view,
  ref_nspace.nspname AS ref_schema
FROM views
  JOIN pg_depend as dep
    ON dep.refobjid = views.view_oid
  JOIN pg_class AS class
    ON views.ref_object = class.oid
  JOIN  pg_namespace AS ref_nspace
      ON class.relnamespace = ref_nspace.oid
  WHERE class.relkind = 'v'
    AND dep.deptype = 'n'
;
 view | schema | ref_view | ref_schema
------+--------+----------+------------
 v2   | public | v1       | public
 v2a  | mytest | v1       | public
```

Example Data

The output for the example queries is based on these database objects and data.

```
CREATE TABLE t1 (
   id integer PRIMARY KEY,
   val text NOT NULL
);

INSERT INTO t1 VALUES
   (1, 'one'), (2, 'two'), (3, 'three');

CREATE FUNCTION f() RETURNS text
   LANGUAGE sql AS 'SELECT ''suffix''::text';

CREATE VIEW v1 AS
  SELECT max(id) AS id
  FROM t1;

CREATE VIEW v2 AS
  SELECT t1.val
  FROM t1 JOIN v1 USING (id);

CREATE VIEW v3 AS
  SELECT val || f()
  FROM t1;

CREATE VIEW v5 AS
  SELECT f() ;

CREATE SCHEMA mytest ;

CREATE TABLE mytest.tm1 (
   id integer PRIMARY KEY,
   val text NOT NULL
);

INSERT INTO mytest.tm1 VALUES
   (1, 'one'), (2, 'two'), (3, 'three');

CREATE VIEW vm1 AS
  SELECT id FROM mytest.tm1 WHERE id < 3 ;
```

```
CREATE VIEW mytest.vm1 AS
  SELECT id FROM public.t1 WHERE id < 3 ;

CREATE VIEW vm2 AS
  SELECT max(id) AS id
  FROM mytest.tm1;

CREATE VIEW mytest.v2a AS
  SELECT t1.val
  FROM public.t1 JOIN public.v1 USING (id);
```

About View Storage in Greenplum Database

A view is similar to a table, both are relations - that is "something with columns". All such objects are stored in the catalog table *pg_class*. These are the general differences:

- A view has no data files (because it holds no data).
- The value of `pg_class.relkind` for a view is `v` rather than `r`.
- A view has an `ON SELECT` query rewrite rule called `_RETURN`.

  The rewrite rule contains the definition of the view and is stored in the `ev_action` column of the *pg_rewrite* catalog table.

For more technical information about views, see the PostgreSQL documentation about *Views and the Rule System*.

Also, a view definition is *not* stored as a string, but in the form of a query parse tree. Views are parsed when they are created, which has several consequences:

- Object names are resolved during `CREATE VIEW`, so the current setting of `search_path` affects the view definition.
- Objects are referred to by their internal immutable object ID rather than by their name. Consequently, renaming an object or column referenced in a view definition can be performed without dropping the view.
- Greenplum Database can determine exactly which objects are used in the view definition, so it can add dependencies on them.

Note that the way Greenplum Database handles views is quite different from the way Greenplum Database handles functions: function bodies are stored as strings and are not parsed when they are created. Consequently, Greenplum Database does not know on which objects a given function depends.

Where View Dependency Information is Stored

These system catalog tables contain the information used to determine the tables on which a view depends.

- *pg_class* - object information including tables and views. The `relkind` column describes the type of object.
- *pg_depend* - object dependency information for database-specific (non-shared) objects.
- *pg_rewrite* - rewrite rules for tables and views.
- *pg_attribute* - information about table columns.
- *pg_namespace* - information about schemas (namespaces).

It is important to note that there is no direct dependency of a view on the objects it uses: the dependent object is actually the view's rewrite rule. That adds another layer of indirection to view dependency information.

Creating and Managing Materialized Views

Materialized views are similar to views. A materialized view enables you to save a frequently used or complex query, then access the query results in a `SELECT` statement as if they were a table. Materialized views persist the query results in a table-like form. While access to the data stored in a materialized view can be much faster than accessing the underlying tables directly or through a view, the data is not always current.

The materialized view data cannot be directly updated. To refresh the materialized view data, use the `REFRESH MATERIALIZED VIEW` command. The query used to create the materialized view is stored in exactly the same way that a view's query is stored. For example, you can create a materialized view that quickly displays a summary of historical sales data for situations where having incomplete data for the current date would be acceptable.

```
CREATE MATERIALIZED VIEW sales_summary AS
  SELECT seller_no, invoice_date, sum(invoice_amt)::numeric(13,2) as
 sales_amt
    FROM invoice
    WHERE invoice_date < CURRENT_DATE
    GROUP BY seller_no, invoice_date
    ORDER BY seller_no, invoice_date;

CREATE UNIQUE INDEX sales_summary_seller
  ON sales_summary (seller_no, invoice_date);
```

The materialized view might be useful for displaying a graph in the dashboard created for sales people. You could schedule a job to update the summary information each night using this command.

```
REFRESH MATERIALIZED VIEW sales_summary;
```

The information about a materialized view in the Greenplum Database system catalogs is exactly the same as it is for a table or view. A materialized view is a relation, just like a table or a view. When a materialized view is referenced in a query, the data is returned directly from the materialized view, just like from a table. The query in the materialized view definition is only used for populating the materialized view.

If you can tolerate periodic updates of materialized view data, the performance benefit can be substantial.

One use of a materialized view is to allow faster access to data brought in from an external data source such as external table or a foreign data wrapper. Also, you can define indexes on a materialized view, whereas foreign data wrappers do not support indexes; this advantage might not apply for other types of external data access.

If a subquery is associated with a single query, consider using the `WITH` clause of the `SELECT` command instead of creating a seldom-used materialized view.

Creating Materialized Views

The `CREATE MATERIALIZED VIEW` command defines a materialized view based on a query.

```
CREATE MATERIALIZED VIEW us_users AS SELECT u.id, u.name, a.zone FROM users
 u, address a WHERE a.country = 'USA';
```

If a materialized view query contains an `ORDER BY` or `SORT` clause, the clause is ignored when a `SELECT` is performed on the materialized query.

Refreshing or Disabling Materialized Views

The `REFRESH MATERIALIZED VIEW` command updates the materialized view data.

```
REFRESH MATERIALIZED VIEW us_users;
```

With the `WITH NO DATA` clause, the current data is removed, no new data is generated, and the materialized view is left in an unscannable state. An error is returned if a query attempts to access an unscannable materialized view.

```
REFRESH MATERIALIZED VIEW us_users WITH NO DATA;
```

Dropping Materialized Views

The `DROP MATERIALIZED VIEW` command removes a materialized view definition and data. For example:

```
DROP MATERIALIZED VIEW us_users;
```

The `DROP MATERIALIZED VIEW ... CASCADE` command also removes all dependent objects. For example, if another materialized view depends on the materialized view which is about to be dropped, the other materialized view will be dropped as well. Without the `CASCADE` option, the `DROP MATERIALIZED VIEW` command fails.

Distribution and Skew

Greenplum Database relies on even distribution of data across segments.

In an MPP shared nothing environment, overall response time for a query is measured by the completion time for all segments. The system is only as fast as the slowest segment. If the data is skewed, segments with more data will take more time to complete, so every segment must have an approximately equal number of rows and perform approximately the same amount of processing. Poor performance and out of memory conditions may result if one segment has significantly more data to process than other segments.

Optimal distributions are critical when joining large tables together. To perform a join, matching rows must be located together on the same segment. If data is not distributed on the same join column, the rows needed from one of the tables are dynamically redistributed to the other segments. In some cases a broadcast motion, in which each segment sends its individual rows to all other segments, is performed rather than a redistribution motion, where each segment rehashes the data and sends the rows to the appropriate segments according to the hash key.

Local (Co-located) Joins

Using a hash distribution that evenly distributes table rows across all segments and results in local joins can provide substantial performance gains. When joined rows are on the same segment, much of the processing can be accomplished within the segment instance. These are called *local* or *co-located* joins. Local joins minimize data movement; each segment operates independently of the other segments, without network traffic or communications between segments.

To achieve local joins for large tables commonly joined together, distribute the tables on the same column. Local joins require that both sides of a join be distributed on the same columns (and in the same order) *and* that all columns in the distribution clause are used when joining tables. The distribution columns must also be the same data type—although some values with different data types may appear to have the same representation, they are stored differently and hash to different values, so they are stored on different segments.

Data Skew

Data skew may be caused by uneven data distribution due to the wrong choice of distribution keys or single tuple table insert or copy operations. Present at the table level, data skew, is often the root cause of poor query performance and out of memory conditions. Skewed data affects scan (read) performance, but it also affects all other query execution operations, for instance, joins and group by operations.

It is very important to *validate* distributions to *ensure* that data is evenly distributed after the initial load. It is equally important to *continue* to validate distributions after incremental loads.

The following query shows the number of rows per segment as well as the variance from the minimum and maximum numbers of rows:

```
SELECT 'Example Table' AS "Table Name",
    max(c) AS "Max Seg Rows", min(c) AS "Min Seg Rows",
    (max(c)-min(c))*100.0/max(c) AS "Percentage Difference Between Max &
 Min"
FROM (SELECT count(*) c, gp_segment_id FROM facts GROUP BY 2) AS a;
```

The `gp_toolkit` schema has two views that you can use to check for skew.

- The `gp_toolkit.gp_skew_coefficients` view shows data distribution skew by calculating the coefficient of variation (CV) for the data stored on each segment. The `skccoeff` column shows the coefficient of variation (CV), which is calculated as the standard deviation divided by the average. It

takes into account both the average and variability around the average of a data series. The lower the value, the better. Higher values indicate greater data skew.

- The `gp_toolkit.gp_skew_idle_fractions` view shows data distribution skew by calculating the percentage of the system that is idle during a table scan, which is an indicator of computational skew. The `siffraction` column shows the percentage of the system that is idle during a table scan. This is an indicator of uneven data distribution or query processing skew. For example, a value of 0.1 indicates 10% skew, a value of 0.5 indicates 50% skew, and so on. Tables that have more than10% skew should have their distribution policies evaluated.

Considerations for Replicated Tables

When you create a replicated table (with the `CREATE TABLE` clause `DISTRIBUTED REPLICATED`), Greenplum Database distributes every table row to every segment instance. Replicated table data is evenly distributed because every segment has the same rows. A query that uses the `gp_segment_id` system column on a replicated table to verify evenly distributed data, will fail because Greenplum Database does not allow queries to reference replicated tables' system columns.

Processing Skew

Processing skew results when a disproportionate amount of data flows to, and is processed by, one or a few segments. It is often the culprit behind Greenplum Database performance and stability issues. It can happen with operations such join, sort, aggregation, and various OLAP operations. Processing skew happens in flight while a query is executing and is not as easy to detect as data skew.

If single segments are failing, that is, not all segments on a host, it may be a processing skew issue. Identifying processing skew is currently a manual process. First look for spill files. If there is skew, but not enough to cause spill, it will not become a performance issue. If you determine skew exists, then find the query responsible for the skew. Following are the steps and commands to use. (Change names like the host file name passed to `gpssh` accordingly):

1. Find the OID for the database that is to be monitored for skew processing:

```
SELECT oid, datname FROM pg_database;
```

Example output:

```
  oid  |  datname
-------+-----------
 17088 | gpadmin
 10899 | postgres
     1 | template1
 10898 | template0
 38817 | pws
 39682 | gpperfmon
(6 rows)
```

2. Run a `gpssh` command to check file sizes across all of the segment nodes in the system. Replace `<OID>` with the OID of the database from the prior command:

```
[gpadmin@mdw kend]$ gpssh -f ~/hosts -e \
    "du -b /data[1-2]/primary/gpseg*/base/<OID>/pgsql_tmp/*" | \
    grep -v "du -b" | sort | awk -F" " '{ arr[$1] = arr[$1] + $2 ; tot =
 tot + $2 }; END \
    { for ( i in arr ) print "Segment node" i, arr[i], "bytes (" arr[i]/
(1024**3)" GB)"; \
    print "Total", tot, "bytes (" tot/(1024**3)" GB)" }' -
```

Example output:

```
Segment node[sdw1] 2443370457 bytes (2.27557 GB)
```

```
Segment node[sdw2] 1766575328 bytes (1.64525 GB)
Segment node[sdw3] 1761686551 bytes (1.6407 GB)
Segment node[sdw4] 1780301617 bytes (1.65804 GB)
Segment node[sdw5] 1742543599 bytes (1.62287 GB)
Segment node[sdw6] 1830073754 bytes (1.70439 GB)
Segment node[sdw7] 1767310099 bytes (1.64594 GB)
Segment node[sdw8] 1765105802 bytes (1.64388 GB)
Total 14856967207 bytes (13.8366 GB)
```

If there is a *significant and sustained* difference in disk usage, then the queries being executed should be investigated for possible skew (the example output above does not reveal significant skew). In monitoring systems, there will always be some skew, but often it is *transient* and will be *short in duration*.

3. If significant and sustained skew appears, the next task is to identify the offending query.

   The command in the previous step sums up the entire node. This time, find the actual segment directory. You can do this from the master or by logging into the specific node identified in the previous step. Following is an example run from the master.

   This example looks specifically for sort files. Not all spill files or skew situations are caused by sort files, so you will need to customize the command:

```
$ gpssh -f ~/hosts -e
    "ls -l /data[1-2]/primary/gpseg*/base/19979/pgsql_tmp/*"
    | grep -i sort | awk '{sub(/base.*tmp\//, ".../", $10); print $1,$6,
$10}' | sort -k2 -n
```

   Here is output from this command:

```
[sdw1] 288718848
      /data1/primary/gpseg2/.../pgsql_tmp_slice0_sort_17758_0001.0[sdw1]
 291176448
      /data2/primary/gpseg5/.../pgsql_tmp_slice0_sort_17764_0001.0[sdw8]
 924581888
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0010.9[sdw4]
 980582400
      /data1/primary/gpseg18/.../pgsql_tmp_slice10_sort_29425_0001.0[sdw6]
 986447872
      /data2/primary/gpseg35/.../pgsql_tmp_slice10_sort_29602_0001.0...
[sdw5] 999620608
      /data1/primary/gpseg26/.../pgsql_tmp_slice10_sort_28637_0001.0[sdw2]
 999751680
      /data2/primary/gpseg9/.../pgsql_tmp_slice10_sort_3969_0001.0[sdw3]
 1000112128
      /data1/primary/gpseg13/.../pgsql_tmp_slice10_sort_24723_0001.0[sdw5]
 1000898560
      /data2/primary/gpseg28/.../pgsql_tmp_slice10_sort_28641_0001.0...
[sdw8] 1008009216
      /data1/primary/gpseg44/.../pgsql_tmp_slice10_sort_15671_0001.0[sdw5]
 1008566272
      /data1/primary/gpseg24/.../pgsql_tmp_slice10_sort_28633_0001.0[sdw4]
 1009451008
      /data1/primary/gpseg19/.../pgsql_tmp_slice10_sort_29427_0001.0[sdw7]
 1011187712
      /data1/primary/gpseg37/.../pgsql_tmp_slice10_sort_18526_0001.0[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0001.0[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0002.1[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0003.2[sdw8]
 1573741824
```

```
     /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0004.3[sdw8]
1573741824
     /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0005.4[sdw8]
1573741824
     /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0006.5[sdw8]
1573741824
     /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0007.6[sdw8]
1573741824
     /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0008.7[sdw8]
1573741824
     /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0009.8
```

Scanning this output reveals that segment `gpseg45` on host `sdw8` is the culprit, as its sort files are larger than the others in the output.

4. Log in to the offending node with `ssh` and become root. Use the `lsof` command to find the PID for the process that owns one of the sort files:

```
[root@sdw8 ~]# lsof /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0002.1
COMMAND  PID    USER    FD   TYPE DEVICE  SIZE        NODE        NAME
postgres 15673  gpadmin 11u  REG  8,48    1073741824  64424546751 /data2/
primary/gpseg45/base/19979/pgsql_tmp/pgsql_tmp_slice10_sort_15673_0002.1
```

   The PID, `15673`, is also part of the file name, but this may not always be the case.

5. Use the `ps` command with the PID to identify the database and connection information:

```
[root@sdw8 ~]# ps -eaf | grep 15673
gpadmin  15673 27471 28 12:05 ?        00:12:59 postgres: port 40003,
 sbaskin bdw
        172.28.12.250(21813) con699238 seg45 cmd32 slice10 MPPEXEC SELECT
root     29622 29566  0 12:50 pts/16   00:00:00 grep 15673
```

6. On the master, check the `pg_log` log file for the user in the previous command (`sbaskin`), connection (`con699238`, and command (`cmd32`). The line in the log file with these three values *should* be the line that contains the query, but occasionally, the command number may differ slightly. For example, the `ps` output may show `cmd32`, but in the log file it is `cmd34`. If the query is still running, the last query for the user and connection is the offending query.

The remedy for processing skew in almost all cases is to rewrite the query. Creating temporary tables can eliminate skew. Temporary tables can be randomly distributed to force a two-stage aggregation.

Inserting, Updating, and Deleting Data

This section provides information about manipulating data and concurrent access in Greenplum Database.

This topic includes the following subtopics:

About Concurrency Control in Greenplum Database

Greenplum Database and PostgreSQL do not use locks for concurrency control. They maintain data consistency using a multiversion model, Multiversion Concurrency Control (MVCC). MVCC achieves transaction isolation for each database session, and each query transaction sees a snapshot of data. This ensures the transaction sees consistent data that is not affected by other concurrent transactions.

Because MVCC does not use explicit locks for concurrency control, lock contention is minimized and Greenplum Database maintains reasonable performance in multiuser environments. Locks acquired for querying (reading) data do not conflict with locks acquired for writing data.

Greenplum Database provides multiple lock modes to control concurrent access to data in tables. Most Greenplum Database SQL commands automatically acquire the appropriate locks to ensure that referenced tables are not dropped or modified in incompatible ways while a command executes. For applications that cannot adapt easily to MVCC behavior, you can use the `LOCK` command to acquire explicit locks. However, proper use of MVCC generally provides better performance.

Table 42: Lock Modes in Greenplum Database

Lock Mode	Associated SQL Commands	Conflicts With
ACCESS SHARE	`SELECT`	ACCESS EXCLUSIVE
ROW SHARE	`SELECT FOR SHARE, SELECT... FOR UPDATE`	EXCLUSIVE, ACCESS EXCLUSIVE
ROW EXCLUSIVE	`INSERT`, `COPY` See *Note*.	SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE UPDATE EXCLUSIVE	`VACUUM` (without `FULL`), `ANALYZE`	SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE	`CREATE INDEX`	ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE

Lock Mode	Associated SQL Commands	Conflicts With
SHARE ROW EXCLUSIVE		ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
EXCLUSIVE	`DELETE`, `UPDATE`, `SELECT...FOR UPDATE`, `REFRESH MATERIALIZED VIEW CONCURRENTLY` See *Note*.	ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
ACCESS EXCLUSIVE	`ALTER TABLE`, `DROP TABLE`, `TRUNCATE`, `REINDEX`, `CLUSTER`, `REFRESH MATERIALIZED VIEW` (without `CONCURRENTLY`), `VACUUM FULL`	ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE

Note: By default Greenplum Database acquires the more restrictive `EXCLUSIVE` lock (rather than `ROW EXCLUSIVE` in PostgreSQL) for `UPDATE`, `DELETE`, and `SELECT...FOR UPDATE` operations on heap tables. When the Global Deadlock Detector is enabled the lock mode for `UPDATE` and `DELETE` operations on heap tables is `ROW EXCLUSIVE`. See *Global Deadlock Detector*. Greenplum always holds a table-level lock with `SELECT...FOR UPDATE` statements.

Inserting Rows

Use the `INSERT` command to create rows in a table. This command requires the table name and a value for each column in the table; you may optionally specify the column names in any order. If you do not specify column names, list the data values in the order of the columns in the table, separated by commas.

For example, to specify the column names and the values to insert:

```
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

To specify only the values to insert:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

Usually, the data values are literals (constants), but you can also use scalar expressions. For example:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod <
'2016-05-07';
```

You can insert multiple rows in a single command. For example:

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

To insert data into a partitioned table, you specify the root partitioned table, the table created with the `CREATE TABLE` command. You also can specify a leaf child table of the partitioned table in an `INSERT` command. An error is returned if the data is not valid for the specified leaf child table. Specifying a child table that is not a leaf child table in the `INSERT` command is not supported.

To insert large amounts of data, use external tables or the `COPY` command. These load mechanisms are more efficient than `INSERT` for inserting large quantities of rows. See *Loading and Unloading Data* for more information about bulk data loading.

The storage model of append-optimized tables is optimized for bulk data loading. Greenplum does not recommend single row `INSERT` statements for append-optimized tables. For append-optimized tables, Greenplum Database supports a maximum of 127 concurrent `INSERT` transactions into a single append-optimized table.

Updating Existing Rows

The `UPDATE` command updates rows in a table. You can update all rows, a subset of all rows, or individual rows in a table. You can update each column separately without affecting other columns.

To perform an update, you need:

- The name of the table and columns to update
- The new values of the columns
- One or more conditions specifying the row or rows to be updated.

For example, the following command updates all products that have a price of *5* to have a price of *10*:

```
UPDATE products SET price = 10 WHERE price = 5;
```

Using `UPDATE` in Greenplum Database has the following restrictions:

- While GPORCA supports updates to Greenplum distribution key columns, the Postgres Planner does not.
- If mirrors are enabled, you cannot use `STABLE` or `VOLATILE` functions in an `UPDATE` statement.
- Greenplum Database partitioning columns cannot be updated.

Deleting Rows

The `DELETE` command deletes rows from a table. Specify a `WHERE` clause to delete rows that match certain criteria. If you do not specify a `WHERE` clause, all rows in the table are deleted. The result is a valid, but empty, table. For example, to remove all rows from the products table that have a price of *10*:

```
DELETE FROM products WHERE price = 10;
```

To delete all rows from a table:

```
DELETE FROM products;
```

Using `DELETE` in Greenplum Database has similar restrictions to using `UPDATE`:

- If mirrors are enabled, you cannot use `STABLE` or `VOLATILE` functions in an `UPDATE` statement.

Truncating a Table

Use the `TRUNCATE` command to quickly remove all rows in a table. For example:

```
TRUNCATE mytable;
```

This command empties a table of all rows in one operation. Note that `TRUNCATE` does not scan the table, therefore it does not process inherited child tables or `ON DELETE` rewrite rules. The command truncates only rows in the named table.

Working With Transactions

Transactions allow you to bundle multiple SQL statements in one all-or-nothing operation.

The following are the Greenplum Database SQL transaction commands:

- BEGIN or START TRANSACTION starts a transaction block.
- END or COMMIT commits the results of a transaction.
- ROLLBACK abandons a transaction without making any changes.
- SAVEPOINT marks a place in a transaction and enables partial rollback. You can roll back commands executed after a savepoint while maintaining commands executed before the savepoint.
- ROLLBACK TO SAVEPOINT rolls back a transaction to a savepoint.
- RELEASE SAVEPOINT destroys a savepoint within a transaction.

Transaction Isolation Levels

Greenplum Database accepts the standard SQL transaction levels as follows:

- READ UNCOMMITTED and READ COMMITTED behave like the standard READ COMMITTED.
- REPEATABLE READ and SERIALIZABLE behave like REPEATABLE READ.

The following information describes the behavior of the Greenplum transaction levels.

Read Uncommitted and Read Committed

Greenplum Database does not allow any command to see an uncommitted update in another concurrent transaction, so READ UNCOMMITTED behaves the same as READ COMMITTED. READ COMMITTED provides fast, simple, partial transaction isolation. SELECT, UPDATE, and DELETE commands operate on a snapshot of the database taken when the query started.

A SELECT query:

- Sees data committed before the query starts.
- Sees updates executed within the transaction.
- Does not see uncommitted data outside the transaction.
- Can possibly see changes that concurrent transactions made if the concurrent transaction is committed after the initial read in its own transaction.

Successive SELECT queries in the same transaction can see different data if other concurrent transactions commit changes between the successive queries. UPDATE and DELETE commands find only rows committed before the commands started.

READ COMMITTED transaction isolation allows concurrent transactions to modify or lock a row before UPDATE or DELETE find the row. READ COMMITTED transaction isolation may be inadequate for applications that perform complex queries and updates and require a consistent view of the database.

Repeatable Read and Serializable

SERIALIZABLE transaction isolation, as defined by the SQL standard, ensures that transactions that run concurrently produce the same results as if they were run one after another. If you specify SERIALIZABLE Greenplum Database falls back to REPEATABLE READ. REPEATABLE READ transactions prevent dirty reads, non-repeatable reads, and phantom reads without expensive locking, but Greenplum Database does not detect all serializability interactions that can occur during concurrent transaction execution. Concurrent transactions should be examined to identify interactions that are not prevented by disallowing concurrent updates of the same data. You can prevent these interactions by using explicit table locks or by requiring the conflicting transactions to update a dummy row introduced to represent the conflict.

With REPEATABLE READ transactions, a SELECT query:

- Sees a snapshot of the data as of the start of the transaction (not as of the start of the current query within the transaction).
- Sees only data committed before the query starts.
- Sees updates executed within the transaction.
- Does not see uncommitted data outside the transaction.
- Does not see changes that concurrent transactions make.

- Successive SELECT commands within a single transaction always see the same data.
- UPDATE, DELETE, SELECT FOR UPDATE, and SELECT FOR SHARE commands find only rows committed before the command started. If a concurrent transaction has updated, deleted, or locked a target row, the REPEATABLE READ transaction waits for the concurrent transaction to commit or roll back the change. If the concurrent transaction commits the change, the REPEATABLE READ transaction rolls back. If the concurrent transaction rolls back its change, theREPEATABLE READ transaction can commit its changes.

The default transaction isolation level in Greenplum Database is READ COMMITTED. To change the isolation level for a transaction, declare the isolation level when you BEGIN the transaction or use the SET TRANSACTION command after the transaction starts.

Global Deadlock Detector

The Greenplum Database Global Deadlock Detector background worker process collects lock information on all segments and uses a directed algorithm to detect the existence of local and global deadlocks. This algorithm allows Greenplum Database to relax concurrent update and delete restrictions on heap tables. (Greenplum Database still employs table-level locking on AO/CO tables, restricting concurrent UPDATE, DELETE, and SELECT...FOR UPDATE operations.)

By default, the Global Deadlock Detector is disabled and Greenplum Database executes the concurrent update and delete operations on a heap table serially. You can enable these concurrent updates and have the Global Deadlock Detector determine when a deadlock exists by setting the server configuration parameter *gp_enable_global_deadlock_detector*.

When the Global Deadlock Detector is enabled, the background worker process is automatically started on the master host when you start Greenplum Database. You configure the interval at which the Global Deadlock Detector collects and analyzes lock waiting data via the *gp_global_deadlock_detector_period* server configuration parameter.

If the Global Deadlock Detector determines that deadlock exists, it breaks the deadlock by cancelling one or more backend processes associated with the youngest transaction(s) involved.

When the Global Deadlock Detector determines a deadlock exists for the following types of transactions, only one of the transactions will succeed. The other transactions will fail with an error indicating that concurrent updates to the same row is not allowed.

- Concurrent transactions on the same row of a heap table where the first transaction is an update operation and a later transaction executes an update or delete and the query plan contains a motion operator.
- Concurrent update transactions on the same distribution key of a heap table that are executed by the Postgres Planner.
- Concurrent update transactions on the same row of a hash table that are executed by the GPORCA optimizer.

  Note: Greenplum Database uses the interval specified in the *deadlock_timeout* server configuration parameter for local deadlock detection. Because the local and global deadlock detection algorithms differ, the cancelled process(es) may differ depending upon which detector (local or global) Greenplum Database triggers first.

  Note: If the *lock_timeout* server configuration parameter is turned on and set to a value smaller than deadlock_timeout and gp_global_deadlock_detector_period, Greenplum Database will abort a statement before it would ever trigger a deadlock check in that session.

To view lock waiting information for all segments, run the gp_dist_wait_status() user-defined function. You can use the output of this function to determine which transactions are waiting on locks, which transactions are holding locks, the lock types and mode, the waiter and holder session identifiers,

and which segments are executing the transactions. Sample output of the `gp_dist_wait_status()` function follows:

```
SELECT * FROM pg_catalog.gp_dist_wait_status();
-[ RECORD 1 ]----+--------------
segid            | 0
waiter_dxid      | 11
holder_dxid      | 12
holdTillEndXact  | t
waiter_lpid      | 31249
holder_lpid      | 31458
waiter_lockmode  | ShareLock
waiter_locktype  | transactionid
waiter_sessionid | 8
holder_sessionid | 9
-[ RECORD 2 ]----+--------------
segid            | 1
waiter_dxid      | 12
holder_dxid      | 11
holdTillEndXact  | t
waiter_lpid      | 31467
holder_lpid      | 31250
waiter_lockmode  | ShareLock
waiter_locktype  | transactionid
waiter_sessionid | 9
holder_sessionid | 8
```

When it cancels a transaction to break a deadlock, the Global Deadlock Detector reports the following error message:

```
ERROR:  canceling statement due to user request: "cancelled by global
 deadlock detector"
```

Global Deadlock Detector UPDATE and DELETE Compatibility

The Global Deadlock Detector can manage concurrent updates for these types of `UPDATE` and `DELETE` commands on heap tables:

- Simple `UPDATE` of a single table. Update a non-distribution key with the Postgres Planner. The command does not contain a `FROM` clause, or a sub-query in the `WHERE` clause.

  ```
  UPDATE t SET c2 = c2 + 1 WHERE c1 > 10;
  ```

- Simple `DELETE` of a single table. The command does not contain a sub-query in the `FROM` or `WHERE` clauses.

  ```
  DELETE FROM t WHERE c1 > 10;
  ```

- Split `UPDATE`. For the Postgres Planner, the `UPDATE` command updates a distribution key.

  ```
  UPDATE t SET c = c + 1; -- c is a distribution key
  ```

  For GPORCA, the `UPDATE` command updates a distribution key or references a distribution key.

  ```
  UPDATE t SET b = b + 1 WHERE c = 10; -- c is a distribution key
  ```

- Complex `UPDATE`. The `UPDATE` command includes multiple table joins.

  ```
  UPDATE t1 SET c = t1.c+1 FROM t2 WHERE t1.c = t2.c;
  ```

Or the command contains a sub-query in the `WHERE` clause.

```
UPDATE t SET c = c + 1 WHERE c > ALL(SELECT * FROM t1);
```

- Complex `DELETE`. A complex `DELETE` command is similar to a complex `UPDATE`, and involves multiple table joins or a sub-query.

```
DELETE FROM t USING t1 WHERE t.c > t1.c;
```

The following table shows the concurrent `UPDATE` or `DELETE` commands that are manged by the Global Deadlock Detector. For example, concurrent simple `UPDATE` commands on the same table row are managed by the Global Deadlock Detector. For a concurrent complex `UPDATE` and a simple `UPDATE`, only one `UPDATE` is performed, and an error is returned for the other `UPDATE`.

Table 43: Concurrent Updates and Deletes Managed by Global Deadlock Detector

	Simple `UPDATE`	Simple `DELETE`	Split `UPDATE`	Complex `UPDATE`	Complex `DELETE`
Simple `UPDATE`	YES	YES	NO	NO	NO
Simple `DELETE`	YES	YES	NO	YES	YES
Split `UPDATE`	NO	NO	NO	NO	NO
Complex `UPDATE`	NO	YES	NO	NO	NO
Complex `DELETE`	NO	YES	NO	NO	YES

Vacuuming the Database

Deleted or updated data rows occupy physical space on disk even though new transactions cannot see them. Periodically running the `VACUUM` command removes these expired rows. For example:

```
VACUUM mytable;
```

The `VACUUM` command collects table-level statistics such as the number of rows and pages. Vacuum all tables after loading data, including append-optimized tables. For information about recommended routine vacuum operations, see *Routine Vacuum and Analyze*.

> **Important:** The `VACUUM`, `VACUUM FULL`, and `VACUUM ANALYZE` commands should be used to maintain the data in a Greenplum database especially if updates and deletes are frequently performed on your database data. See the `VACUUM` command in the *Greenplum Database Reference Guide* for information about using the command.

Running Out of Locks

Greenplum Database can potentially run out of locks when a database operation accesses multiple tables in a single transaction. Backup and restore are examples of such operations.

When Greenplum Database runs out of locks, the error message that you may observe references a shared memory error:

```
... "WARNING","53200","out of shared memory",,,,,,"LOCK TABLE ...
... "ERROR","53200","out of shared memory",,"You might need to increase
 max_locks_per_transaction.",,,,"LOCK TABLE ...
```

> **Note:** "shared memory" in this context refers to the shared memory of the internal object: the lock slots. "Out of shared memory" does *not* refer to exhaustion of system- or Greenplum-level memory resources.

As the hint describes, consider increasing the `max_locks_per_transaction` server configuration parameter when you encounter this error.

Querying Data

This topic provides information about using SQL in Greenplum databases.

You enter SQL statements called queries to view, change, and analyze data in a database using the `psql` interactive SQL client and other client tools.

About Greenplum Query Processing

This topic provides an overview of how Greenplum Database processes queries. Understanding this process can be useful when writing and tuning queries.

Users issue queries to Greenplum Database as they would to any database management system. They connect to the database instance on the Greenplum master host using a client application such as `psql` and submit SQL statements.

Understanding Query Planning and Dispatch

The master receives, parses, and optimizes the query. The resulting query plan is either parallel or targeted. The master dispatches parallel query plans to all segments, as shown in *Figure 21: Dispatching the Parallel Query Plan*. The master dispatches targeted query plans to a single segment, as shown in *Figure 22: Dispatching a Targeted Query Plan*. Each segment is responsible for executing local database operations on its own set of data.

Most database operations—such as table scans, joins, aggregations, and sorts—execute across all segments in parallel. Each operation is performed on a segment database independent of the data stored in the other segment databases.

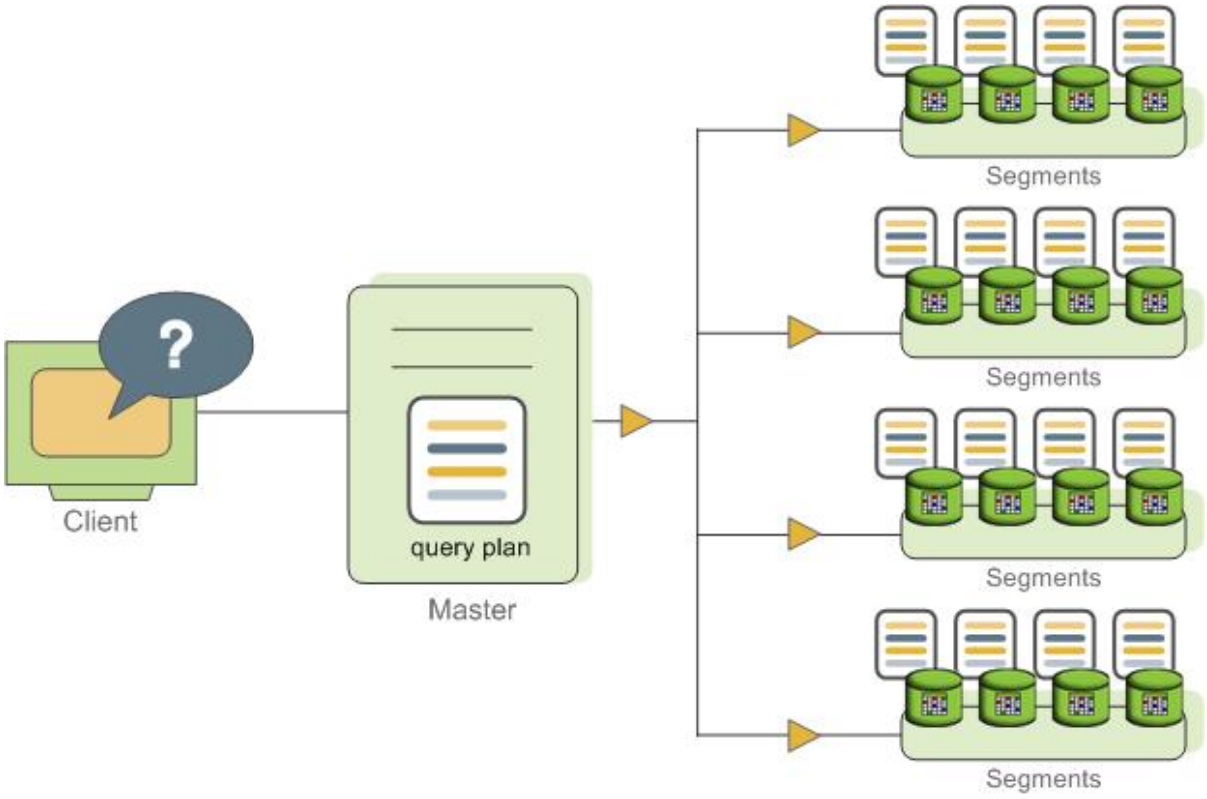


Figure 21: Dispatching the Parallel Query Plan

Certain queries may access only data on a single segment, such as single-row `INSERT`, `UPDATE`, `DELETE`, or `SELECT` operations or queries that filter on the table distribution key column(s). In queries such as these, the query plan is not dispatched to all segments, but is targeted at the segment that contains the affected or relevant row(s).

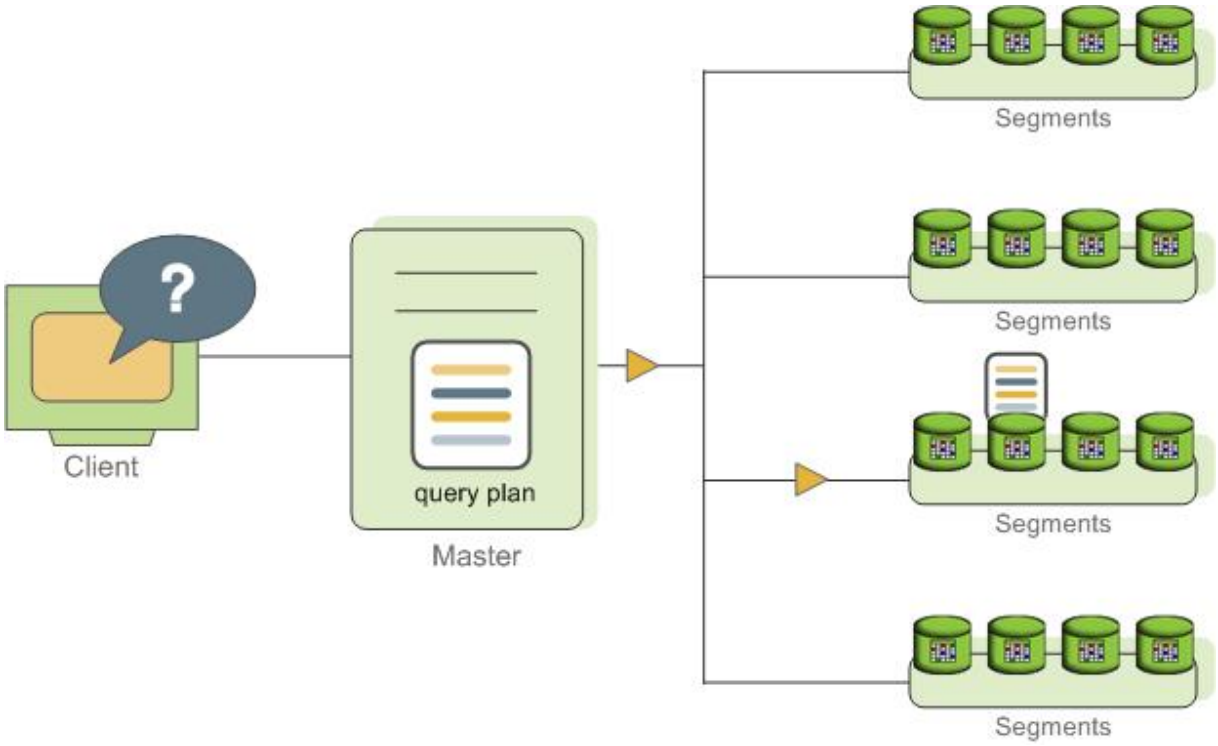


Figure 22: Dispatching a Targeted Query Plan

Understanding Greenplum Query Plans

A query plan is the set of operations Greenplum Database will perform to produce the answer to a query. Each *node* or step in the plan represents a database operation such as a table scan, join, aggregation, or sort. Plans are read and executed from bottom to top.

In addition to common database operations such as table scans, joins, and so on, Greenplum Database has an additional operation type called *motion*. A motion operation involves moving tuples between the segments during query processing. Note that not every query requires a motion. For example, a targeted query plan does not require data to move across the interconnect.

To achieve maximum parallelism during query execution, Greenplum divides the work of the query plan into *slices*. A slice is a portion of the plan that segments can work on independently. A query plan is sliced wherever a *motion* operation occurs in the plan, with one slice on each side of the motion.

For example, consider the following simple query involving a join between two tables:

```
SELECT customer, amount
FROM sales JOIN customer USING (cust_id)
WHERE dateCol = '04-30-2016';
```

Figure 23: Query Slice Plan shows the query plan. Each segment receives a copy of the query plan and works on it in parallel.

The query plan for this example has a *redistribute motion* that moves tuples between the segments to complete the join. The redistribute motion is necessary because the customer table is distributed across the segments by `cust_id`, but the sales table is distributed across the segments by `sale_id`. To perform

the join, the `sales` tuples must be redistributed by `cust_id`. The plan is sliced on either side of the redistribute motion, creating *slice 1* and *slice 2*.

This query plan has another type of motion operation called a *gather motion*. A gather motion is when the segments send results back up to the master for presentation to the client. Because a query plan is always sliced wherever a motion occurs, this plan also has an implicit slice at the very top of the plan (*slice 3*). Not all query plans involve a gather motion. For example, a `CREATE TABLE x AS SELECT...` statement would not have a gather motion because tuples are sent to the newly created table, not to the master.

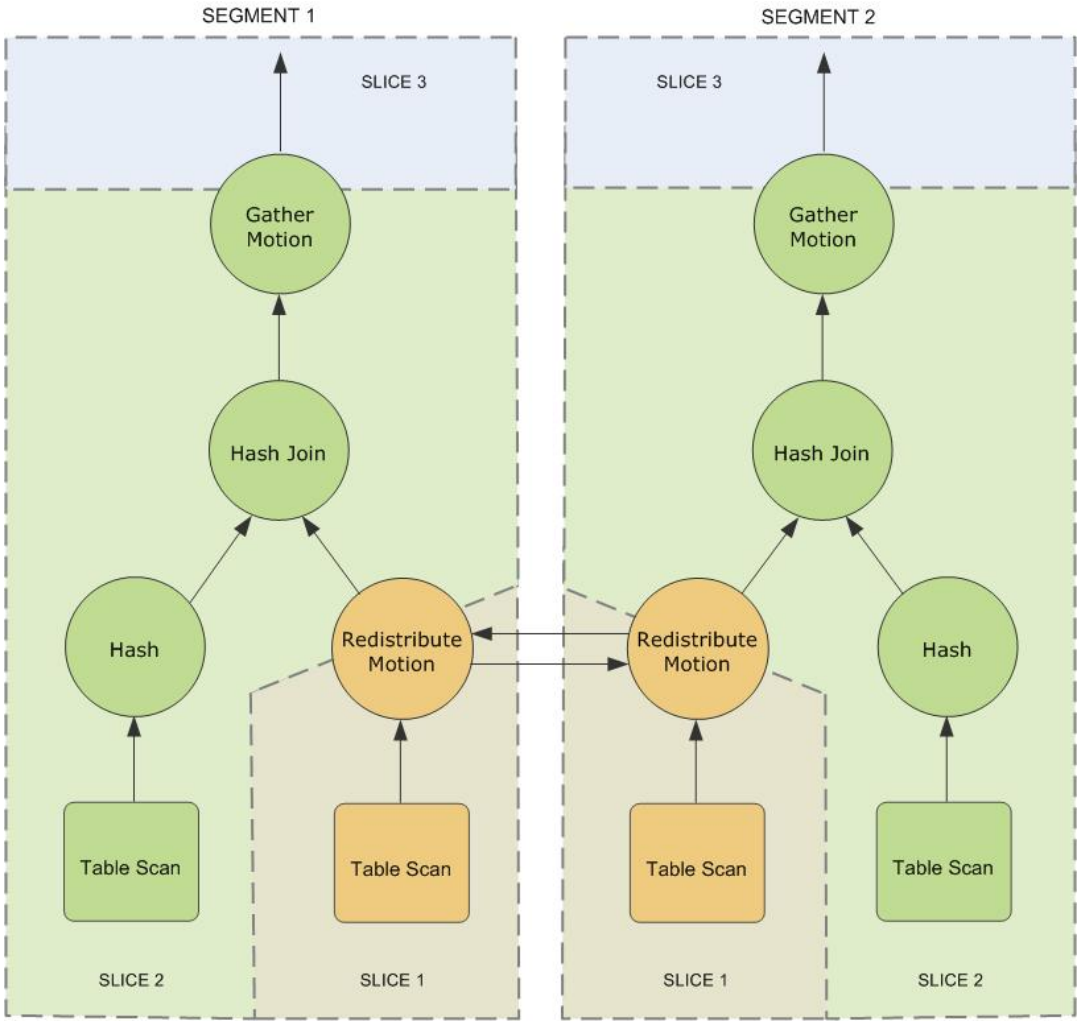


Figure 23: Query Slice Plan

Understanding Parallel Query Execution

Greenplum creates a number of database processes to handle the work of a query. On the master, the query worker process is called the *query dispatcher* (QD). The QD is responsible for creating and dispatching the query plan. It also accumulates and presents the final results. On the segments, a query worker process is called a *query executor* (QE). A QE is responsible for completing its portion of work and communicating its intermediate results to the other worker processes.

There is at least one worker process assigned to each *slice* of the query plan. A worker process works on its assigned portion of the query plan independently. During query execution, each segment will have a number of processes working on the query in parallel.

Related processes that are working on the same slice of the query plan but on different segments are called *gangs*. As a portion of work is completed, tuples flow up the query plan from one gang of processes

to the next. This inter-process communication between the segments is referred to as the *interconnect* component of Greenplum Database.

Figure 24: Query Worker Processes shows the query worker processes on the master and two segment instances for the query plan illustrated in *Figure 23: Query Slice Plan*.

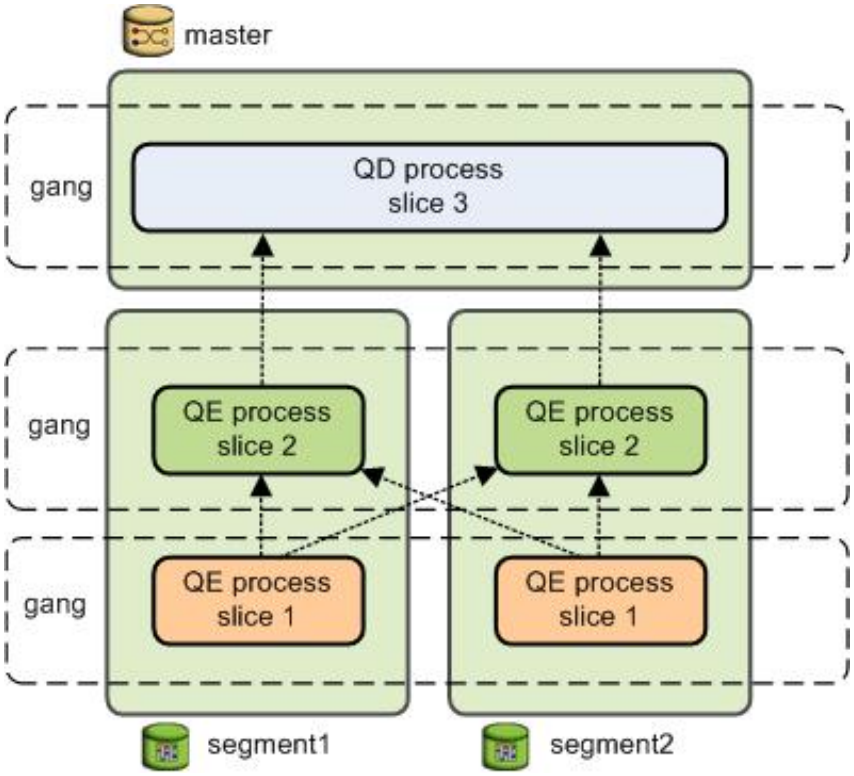


Figure 24: Query Worker Processes

About GPORCA

In Greenplum Database, the default GPORCA optmizer co-exists with the Postgres Planner.

These sections describe GPORCA functionality and usage:

- *Overview of GPORCA*
- *Enabling and Disabling GPORCA*
- *Considerations when Using GPORCA*
- *GPORCA Features and Enhancements*
- *Changed Behavior with the GPORCA*
- *GPORCA Limitations*
- *Determining the Query Optimizer that is Used*
- *About Uniform Multi-level Partitioned Tables*

Overview of GPORCA

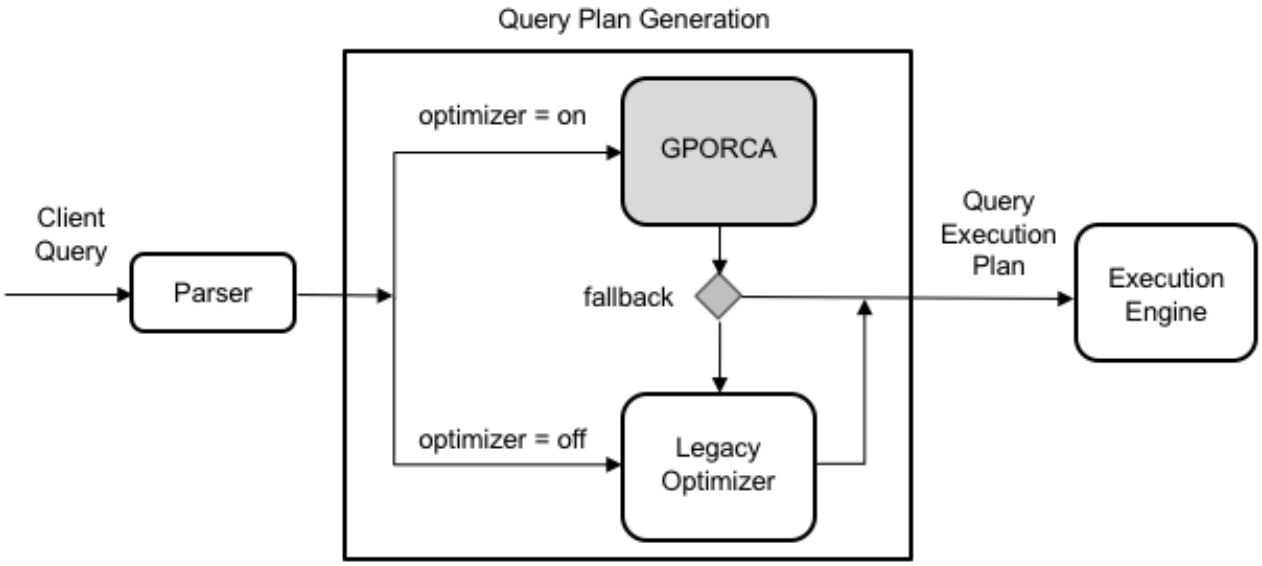
GPORCA extends the planning and optimization capabilities of the Postgres Planner. GPORCA is extensible and achieves better optimization in multi-core architecture environments. Greenplum Database uses GPORCA by default to generate an execution plan for a query when possible.

GPORCA also enhances Greenplum Database query performance tuning in the following areas:

- Queries against partitioned tables
- Queries that contain a common table expression (CTE)
- Queries that contain subqueries

In Greenplum Database, GPORCA co-exists with the Postgres Planner. By default, Greenplum Database uses GPORCA. If GPORCA cannot be used, then the Postgres Planner is used.

The following figure shows how GPORCA fits into the query planning architecture.



Note: All Postgres Planner server configuration parameters are ignored by GPORCA. However, if Greenplum Database falls back to the Postgres Planner, the planner server configuration parameters will impact the query plan generation. For a list of Postgres Planner server configuration parameters, see *Query Tuning Parameters*.

Enabling and Disabling GPORCA

By default, Greenplum Database uses GPORCA instead of the Postgres Planner. Server configuration parameters enable or disable GPORCA.

Although GPORCA is on by default, you can configure GPORCA usage at the system, database, session, or query level using the `optimizer` parameter. Refer to one of the following sections if you want to change the default behavior:

- *Enabling GPORCA for a System*
- *Enabling GPORCA for a Database*
- *Enabling GPORCA for a Session or a Query*

Note: You can disable the ability to enable or disable GPORCA with the server configuration parameter `optimizer_control`. For information about the server configuration parameters, see the *Greenplum Database Reference Guide*.

Enabling GPORCA for a System

Set the server configuration parameter `optimizer` for the Greenplum Database system.

1. Log into the Greenplum Database master host as `gpadmin`, the Greenplum Database administrator.
2. Set the values of the server configuration parameters. These Greenplum Database `gpconfig` utility commands sets the value of the parameters to on:

```
$ gpconfig -c optimizer -v on --masteronly
```

3. Restart Greenplum Database. This Greenplum Database `gpstop` utility command reloads the `postgresql.conf` files of the master and segments without shutting down Greenplum Database.

```
gpstop -u
```

Enabling GPORCA for a Database

Set the server configuration parameter `optimizer` for individual Greenplum databases with the `ALTER DATABASE` command. For example, this command enables GPORCA for the database *test_db*.

```
> ALTER DATABASE test_db SET OPTIMIZER = ON ;
```

Enabling GPORCA for a Session or a Query

You can use the `SET` command to set `optimizer` server configuration parameter for a session. For example, after you use the `psql` utility to connect to Greenplum Database, this `SET` command enables GPORCA:

```
> set optimizer = on ;
```

To set the parameter for a specific query, include the `SET` command prior to running the query.

Collecting Root Partition Statistics

For a partitioned table, GPORCA uses statistics of the table root partition to generate query plans. These statistics are used for determining the join order, for splitting and joining aggregate nodes, and for costing the query steps. In contrast, the Postgres Planner uses the statistics of each leaf partition.

If you execute queries on partitioned tables, you should collect statistics on the root partition and periodically update those statistics to ensure that GPORCA can generate optimal query plans. If the root partition statistics are not up-to-date or do not exist, GPORCA still performs dynamic partition elimination for queries against the table. However, the query plan might not be optimal.

Running ANALYZE

By default, running the `ANALYZE` command on the root partition of a partitioned table samples the leaf partition data in the table, and stores the statistics for the root partition. `ANALYZE` collects statistics on the root and leaf partitions, including HyperLogLog (HLL) statistics on the leaf partitions. `ANALYZE ROOTPARTITION` collects statistics only on the root partition. The server configuration parameter *`optimizer_analyze_root_partition`* controls whether the `ROOTPARTITION` keyword is required to collect root statistics for the root partition of a partitioned table. See the *`ANALYZE`* command for information about collecting statistics on partitioned tables.

Keep in mind that `ANALYZE` always scans the entire table before updating the root partition statistics. If your table is very large, this operation can take a significant amount of time. `ANALYZE ROOTPARTITION` also uses an `ACCESS SHARE` lock that prevents certain operations, such as `TRUNCATE` and `VACUUM` operations, during execution. For these reasons, you should schedule `ANALYZE` operations periodically, or when there are significant changes to leaf partition data.

Follow these best practices for running `ANALYZE` or `ANALYZE ROOTPARTITION` on partitioned tables in your system:

- Run `ANALYZE <`*`root_partition`*`>` on a new partitioned table after adding initial data. Run `ANALYZE <`*`leaf_partition`*`>` on a new leaf partition or a leaf partition where data has changed. By default, running the command on a leaf partition updates the root partition statistics if the other leaf partitions have statistics.
- Update root partition statistics when you observe query performance regression in `EXPLAIN` plans against the table, or after significant changes to leaf partition data. For example, if you add a new leaf partition at some point after generating root partition statistics, consider running `ANALYZE` or `ANALYZE`

ROOTPARTITION to update root partition statistics with the new tuples inserted from the new leaf partition.
- For very large tables, run ANALYZE or ANALYZE ROOTPARTITION only weekly, or at some interval longer than daily.
- Avoid running ANALYZE with no arguments, because doing so executes the command on all database tables including partitioned tables. With large databases, these global ANALYZE operations are difficult to monitor, and it can be difficult to predict the time needed for completion.
- Consider running multiple ANALYZE *<table_name>* or ANALYZE ROOTPARTITION *<table_name>* operations in parallel to speed the operation of statistics collection, if your I/O throughput can support the load.
- You can also use the Greenplum Database utility analyzedb to update table statistics. Using analyzedb ensures that tables that were previously analzyed are not re-analyzed if no modifications were made to the leaf partition.

GPORCA and Leaf Partition Statistics

Although creating and maintaining root partition statistics is crucial for GPORCA query performance with partitioned tables, maintaining leaf partition statistics is also important. If GPORCA cannot generate a plan for a query against a partitioned table, then the Postgres Planner is used and leaf partition statistics are needed to produce the optimal plan for that query.

GPORCA itself also uses leaf partition statistics for any queries that access leaf partitions directly, instead of using the root partition with predicates to eliminate partitions. For example, if you know which partitions hold necessary tuples for a query, you can directly query the leaf partition table itself; in this case GPORCA uses the leaf partition statistics.

Disabling Automatic Root Partition Statistics Collection

If you do not intend to execute queries on partitioned tables with GPORCA (setting the server configuration parameter *optimizer* to off), then you can disable the automatic collection of statistics on the root partition of the partitioned table. The server configuration parameter *optimizer_analyze_root_partition* controls whether the ROOTPARTITION keyword is required to collect root statistics for the root partition of a partitioned table. The default setting for the parameter is on, the ANALYZE command can collect root partition statistics without the ROOTPARTITION keyword. You can disable automatic collection of root partition statistics by setting the parameter to off. When the value is off, you must run ANALZYE ROOTPARTITION to collect root partition statistics.

1. Log into the Greenplum Database master host as gpadmin, the Greenplum Database administrator.
2. Set the values of the server configuration parameters. These Greenplum Database gpconfig utility commands sets the value of the parameters to off:

```
$ gpconfig -c optimizer_analyze_root_partition -v off --masteronly
```

3. Restart Greenplum Database. This Greenplum Database gpstop utility command reloads the postgresql.conf files of the master and segments without shutting down Greenplum Database.

```
gpstop -u
```

Considerations when Using GPORCA

To execute queries optimally with GPORCA, query criteria to consider.

Ensure the following criteria are met:

- The table does not contain multi-column partition keys.
- The multi-level partitioned table is a uniform multi-level partitioned table. See *About Uniform Multi-level Partitioned Tables*.

- The server configuration parameter `optimizer_enable_master_only_queries` is set to `on` when running against master only tables such as the system table *pg_attribute*. For information about the parameter, see the *Greenplum Database Reference Guide*.

  Note: Enabling this parameter decreases performance of short running catalog queries. To avoid this issue, set this parameter only for a session or a query.
- Statistics have been collected on the root partition of a partitioned table.

If the partitioned table contains more than 20,000 partitions, consider a redesign of the table schema.

These server configuration parameters affect GPORCA query processing.

- `optimizer_cte_inlining_bound` controls the amount of inlining performed for common table expression (CTE) queries (queries that contain a `WHERE` clause).
- `optimizer_force_multistage_agg` forces GPORCA to choose a multi-stage aggregate plan for a scalar distinct qualified aggregate. When the value is `off` (the default), GPORCA chooses between a one-stage and two-stage aggregate plan based on cost.
- `optimizer_force_three_stage_scalar_dqa` forces GPORCA to choose a plan with multistage aggregates when such a plan alternative is generated.
- `optimizer_join_order` sets the query optimization level for join ordering by specifying which types of join ordering alternatives to evaluate.
- `optimizer_join_order_threshold` specifies the maximum number of join children for which GPORCA uses the dynamic programming-based join ordering algorithm.
- `optimizer_nestloop_factor` controls nested loop join cost factor to apply to during query optimization.
- `optimizer_parallel_union` controls the amount of parallelization that occurs for queries that contain a `UNION` or `UNION ALL` clause. When the value is `on`, GPORCA can generate a query plan the child operations of a `UNION` or `UNION ALL` operation execute in parallel on segment instances.
- `optimizer_sort_factor` controls the cost factor that GPORCA applies to sorting operations during query optimization. The cost factor can be adjusted for queries when data skew is present.
- `gp_enable_relsize_collection` controls how GPORCA (and the Postgres Planner) handle a table without statistics. By default, GPORCA uses a default value to estimate the number of rows if statistics are not available. When this value is `on`, GPORCA uses the estimated size of a table if there are no statistics for the table.

  This parameter is ignored for a root partition of a partitioned table. If the root partition does not have statistics, GPORCA always uses the default value. You can use `ANALZYE ROOTPARTITION` to collect statistics on the root partition. See *ANALYZE*.

These server configuration parameters control the display and logging of information.

- `optimizer_print_missing_stats` controls the display of column information about columns with missing statistics for a query (default is `true`)
- `optimizer_print_optimization_stats` controls the logging of GPORCA query optimization metrics for a query (default is `off`)

For information about the parameters, see the *Greenplum Database Reference Guide*.

GPORCA generates minidumps to describe the optimization context for a given query. The minidump files are used by Pivotal support to analyze Greenplum Database issues. The information in the file is not in a format that can be easily used for debugging or troubleshooting. The minidump file is located under the master data directory and uses the following naming format:

`Minidump_`*`date_time`*`.mdp`

For information about the minidump file, see the server configuration parameter `optimizer_minidump` in the *Greenplum Database Reference Guide*.

When the `EXPLAIN ANALYZE` command uses GPORCA, the `EXPLAIN` plan shows only the number of partitions that are being eliminated. The scanned partitions are not shown. To show

the name of the scanned partitions in the segment logs set the server configuration parameter `gp_log_dynamic_partition_pruning` to `on`. This example `SET` command enables the parameter.

```
SET gp_log_dynamic_partition_pruning = on;
```

GPORCA Features and Enhancements

GPORCA, the Greenplum next generation query optimizer, includes enhancements for specific types of queries and operations:

- *Queries Against Partitioned Tables*
- *Queries that Contain Subqueries*
- *Queries that Contain Common Table Expressions*
- *DML Operation Enhancements with GPORCA*

GPORCA also includes these optimization enhancements:

- Improved join ordering
- Join-Aggregate reordering
- Sort order optimization
- Data skew estimates included in query optimization

Queries Against Partitioned Tables

GPORCA includes these enhancements for queries against partitioned tables:

- Partition elimination is improved.
- Uniform multi-level partitioned tables are supported. For information about uniform multi-level partitioned tables, see *About Uniform Multi-level Partitioned Tables*
- Query plan can contain the `Partition selector` operator.
- Partitions are not enumerated in `EXPLAIN` plans.

  For queries that involve static partition selection where the partitioning key is compared to a constant, GPORCA lists the number of partitions to be scanned in the `EXPLAIN` output under the Partition Selector operator. This example Partition Selector operator shows the filter and number of partitions selected:

  ```
  Partition Selector for Part_Table (dynamic scan id: 1)
         Filter: a > 10
         Partitions selected:  1 (out of 3)
  ```

  For queries that involve dynamic partition selection where the partitioning key is compared to a variable, the number of partitions that are scanned will be known only during query execution. The partitions selected are not shown in the `EXPLAIN` output.
- Plan size is independent of number of partitions.
- Out of memory errors caused by number of partitions are reduced.

This example `CREATE TABLE` command creates a range partitioned table.

```
CREATE TABLE sales(order_id int, item_id int, amount numeric(15,2),
      date date, yr_qtr int)
   range partitioned by yr_qtr;
```

GPORCA improves on these types of queries against partitioned tables:

- Full table scan. Partitions are not enumerated in plans.

  ```
  SELECT * FROM sales;
  ```

- Query with a constant filter predicate. Partition elimination is performed.

```
SELECT * FROM sales WHERE yr_qtr = 201501;
```

- Range selection. Partition elimination is performed.

```
SELECT * FROM sales WHERE yr_qtr BETWEEN 201601 AND 201704 ;
```

- Joins involving partitioned tables. In this example, the partitioned dimension table *date_dim* is joined with fact table *catalog_sales*:

```
SELECT * FROM catalog_sales
   WHERE date_id IN (SELECT id FROM date_dim WHERE month=12);
```

Queries that Contain Subqueries

GPORCA handles subqueries more efficiently. A subquery is query that is nested inside an outer query block. In the following query, the `SELECT` in the `WHERE` clause is a subquery.

```
SELECT * FROM part
  WHERE price > (SELECT avg(price) FROM part);
```

GPORCA also handles queries that contain a correlated subquery (CSQ) more efficiently. A correlated subquery is a subquery that uses values from the outer query. In the following query, the `price` column is used in both the outer query and the subquery.

```
SELECT * FROM part p1
  WHERE price > (SELECT avg(price) FROM part p2
  WHERE  p2.brand = p1.brand);
```

GPORCA generates more efficient plans for the following types of subqueries:

- CSQ in the `SELECT` list.

```
SELECT *,
 (SELECT min(price) FROM part p2 WHERE p1.brand = p2.brand)
 AS foo
FROM part p1;
```

- CSQ in disjunctive (`OR`) filters.

```
SELECT FROM part p1 WHERE p_size > 40 OR
      p_retailprice >
      (SELECT avg(p_retailprice)
          FROM part p2
          WHERE p2.p_brand = p1.p_brand)
```

- Nested CSQ with skip level correlations

```
SELECT * FROM part p1 WHERE p1.p_partkey
IN (SELECT p_partkey FROM part p2 WHERE p2.p_retailprice =
     (SELECT min(p_retailprice)
       FROM part p3
       WHERE p3.p_brand = p1.p_brand)
);
```

  Note: Nested CSQ with skip level correlations are not supported by the Postgres Planner.

- CSQ with aggregate and inequality. This example contains a CSQ with an inequality.

```
SELECT * FROM part p1 WHERE p1.p_retailprice =
```

```
 (SELECT min(p_retailprice) FROM part p2 WHERE p2.p_brand <> p1.p_brand);
```

- CSQ that must return one row.

```
SELECT p_partkey,
  (SELECT p_retailprice FROM part p2 WHERE p2.p_brand = p1.p_brand )
FROM part p1;
```

Queries that Contain Common Table Expressions

GPORCA handles queries that contain the `WITH` clause. The `WITH` clause, also known as a common table expression (CTE), generates temporary tables that exist only for the query. This example query contains a CTE.

```
WITH v AS (SELECT a, sum(b) as s FROM T where c < 10 GROUP BY a)
  SELECT *FROM  v AS v1 ,  v AS v2
  WHERE v1.a <> v2.a AND v1.s < v2.s;
```

As part of query optimization, GPORCA can push down predicates into a CTE. For example query, GPORCA pushes the equality predicates to the CTE.

```
WITH v AS (SELECT a, sum(b) as s FROM T GROUP BY a)
  SELECT *
  FROM v as v1, v as v2, v as v3
  WHERE v1.a < v2.a
    AND v1.s < v3.s
    AND v1.a = 10
    AND v2.a = 20
    AND v3.a = 30;
```

GPORCA can handle these types of CTEs:

- CTE that defines one or multiple tables. In this query, the CTE defines two tables.

```
WITH cte1 AS (SELECT a, sum(b) as s FROM T
                where c < 10 GROUP BY a),
     cte2 AS (SELECT a, s FROM cte1 where s > 1000)
  SELECT *
  FROM cte1 as v1, cte2 as v2, cte2 as v3
  WHERE v1.a < v2.a AND v1.s < v3.s;
```

- Nested CTEs.

```
WITH v AS (WITH w AS (SELECT a, b FROM foo
                      WHERE b < 5)
           SELECT w1.a, w2.b
           FROM w AS w1, w AS w2
           WHERE w1.a = w2.a AND w1.a > 2)
  SELECT v1.a, v2.a, v2.b
  FROM v as v1, v as v2
  WHERE v1.a < v2.a;
```

DML Operation Enhancements with GPORCA

GPORCA contains enhancements for DML operations such as `INSERT`, `UPDATE`, and `DELETE`.

- A DML node in a query plan is a query plan operator.
  - Can appear anywhere in the plan, as a regular node (top slice only for now)
  - Can have consumers
- `UPDATE` operations use the query plan operator `Split` and supports these operations:

- `UPDATE` operations on the table distribution key columns.
- `UPDATE` operations on the table on the partition key column.

This example plan shows the `Split` operator.

```
QUERY PLAN
------------------------------------------------------------------
Update  (cost=0.00..5.46 rows=1 width=1)
   ->  Redistribute Motion 2:2  (slice1; segments: 2)
         Hash Key: a
         ->  Result  (cost=0.00..3.23 rows=1 width=48)
               ->  Split  (cost=0.00..2.13 rows=1 width=40)
                     ->  Result  (cost=0.00..1.05 rows=1 width=40)
                           ->  Seq Scan on dmltest
```

- New query plan operator `Assert` is used for constraints checking.

  This example plan shows the `Assert` operator.

```
QUERY PLAN
------------------------------------------------------------
 Insert  (cost=0.00..4.61 rows=3 width=8)
   ->  Assert  (cost=0.00..3.37 rows=3 width=24)
         Assert Cond: (dmlsource.a > 2) IS DISTINCT FROM
false
         ->  Assert  (cost=0.00..2.25 rows=3 width=24)
               Assert Cond: NOT dmlsource.b IS NULL
               ->  Result  (cost=0.00..1.14 rows=3 width=24)
                     ->  Seq Scan on dmlsource
```

Changed Behavior with the GPORCA

There are changes to Greenplum Database behavior with the GPORCA optimizer enabled (the default) as compared to the Postgres Planner.

- `UPDATE` operations on distribution keys are allowed.
- `UPDATE` operations on partitioned keys are allowed.
- Queries against uniform partitioned tables are supported.
- Queries against partitioned tables that are altered to use an external table as a leaf child partition fall back to the Postgres Planner.
- Except for `INSERT`, DML operations directly on partition (child table) of a partitioned table are not supported.

  For the `INSERT` command, you can specify a leaf child table of the partitioned table to insert data into a partitioned table. An error is returned if the data is not valid for the specified leaf child table. Specifying a child table that is not a leaf child table is not supported.
- The command `CREATE TABLE AS` distributes table data randomly if the `DISTRIBUTED BY` clause is not specified and no primary or unique keys are specified.
- Non-deterministic updates not allowed. The following `UPDATE` command returns an error.

```
update r set b =  r.b  + 1 from s where  r.a  in (select a from s);
```

- Statistics are required on the root table of a partitioned table. The `ANALYZE` command generates statistics on both root and individual partition tables (leaf child tables). See the `ROOTPARTITION` clause for `ANALYZE` command.
- Additional Result nodes in the query plan:
  - Query plan `Assert` operator.
  - Query plan `Partition selector` operator.
  - Query plan `Split` operator.

- When running `EXPLAIN`, the query plan generated by GPORCA is different than the plan generated by the Postgres Planner.
- Greenplum Database adds the log file message `Planner produced plan` when GPORCA is enabled and Greenplum Database falls back to the Postgres Planner to generate the query plan.
- Greenplum Database issues a warning when statistics are missing from one or more table columns. When executing an SQL command with GPORCA, Greenplum Database issues a warning if the command performance could be improved by collecting statistics on a column or set of columns referenced by the command. The warning is issued on the command line and information is added to the Greenplum Database log file. For information about collecting statistics on table columns, see the `ANALYZE` command in the *Greenplum Database Reference Guide*.

GPORCA Limitations

There are limitations in Greenplum Database when using the default GPORCA optimizer. GPORCA and the Postgres Planner currently coexist in Greenplum Database because GPORCA does not support all Greenplum Database features.

This section describes the limitations.

- *Unsupported SQL Query Features*
- *Performance Regressions*

Unsupported SQL Query Features

Certain query features are not supported with the default GPORCA optimizer. When an unsupported query is executed, Greenplum logs this notice along with the query text:

```
Feature not supported by the Pivotal Query Optimizer: UTILITY command
```

These features are unsupported when GPORCA is enabled (the default):

- Prepared statements that have parameterized values.
- Indexed expressions (an index defined as expression based on one or more columns of the table)
- SP-GiST indexing method. GPORCA supports only B-tree, bitmap, GIN, and GiST indexes. GPORCA ignores indexes created with unsupported methods.
- External parameters
- These types of partitioned tables:
  - Non-uniform partitioned tables.
  - Partitioned tables that have been altered to use an external table as a leaf child partition.
- SortMergeJoin (SMJ).
- Ordered aggregations.
- These analytics extensions:
  - CUBE
  - Multiple grouping sets
- These scalar operators:
  - `ROW`
  - `ROWCOMPARE`
  - `FIELDSELECT`
- Aggregate functions that take set operators as input arguments.
- `percentile_*` window functions (ordered-set aggregate functions).
- Inverse distribution functions.
- Queries that execute functions that are defined with the `ON MASTER` or `ON ALL SEGMENTS` attribute.
- Queries that contain UNICODE characters in metadata names, such as table names, and the characters are not compatible with the host system locale.

- `SELECT`, `UPDATE`, and `DELETE` commands where a table name is qualified by the `ONLY` keyword.
- Per-column collation. GPORCA supports collation only when all columns in the query use the same collation. If columns in the query use different collations, then Greenplum uses the Postgres Planner.

Performance Regressions

The following features are known performance regressions that occur with GPORCA enabled:

- Short running queries - For GPORCA, short running queries might encounter additional overhead due to GPORCA enhancements for determining an optimal query execution plan.
- `ANALYZE` - For GPORCA, the `ANALYZE` command generates root partition statistics for partitioned tables. For the Postgres Planner, these statistics are not generated.
- DML operations - For GPORCA, DML enhancements including the support of updates on partition and distribution keys might require additional overhead.

Also, enhanced functionality of the features from previous versions could result in additional time required when GPORCA executes SQL statements with the features.

Determining the Query Optimizer that is Used

When GPORCA is enabled (the default), you can determine if Greenplum Database is using GPORCA or is falling back to the Postgres Planner.

You can examine the `EXPLAIN` query plan for the query determine which query optimizer was used by Greenplum Database to execute the query:

- When GPORCA generates the query plan, the setting `optimizer=on` and GPORCA version are displayed at the end of the query plan. For example.

  ```
  Settings:  optimizer=on
  Optimizer status: Pivotal Optimizer (GPORCA) version 1.584
  ```

  When Greenplum Database falls back to the Postgres Planner to generate the plan, the setting `optimizer=on` and `Postgres query optimizer` are displayed at the end of the query plan. For example.

  ```
  Settings:  optimizer=on
  Optimizer status: Postgres query optimizer
  ```

  When the server configuration parameter `OPTIMIZER` is `off`, these lines are displayed at the end of a query plan.

  ```
  Settings:  optimizer=off
  Optimizer status: Postgres query optimizer
  ```

- These plan items appear only in the `EXPLAIN` plan output generated by GPORCA. The items are not supported in a Postgres Planner query plan.
  - Assert operator
  - Sequence operator
  - DynamicIndexScan
  - DynamicSeqScan
- When a query against a partitioned table is generated by GPORCA, the `EXPLAIN` plan displays only the number of partitions that are being eliminated is listed. The scanned partitions are not shown. The `EXPLAIN` plan generated by the Postgres Planner lists the scanned partitions.

The log file contains messages that indicate which query optimizer was used. If Greenplum Database falls back to the Postgres Planner, a message with `NOTICE` information is added to the log file that indicates the unsupported feature. Also, the label `Planner produced plan:` appears before the query in the query execution log message when Greenplum Database falls back to the Postgres optimizer.

Note: You can configure Greenplum Database to display log messages on the psql command line by setting the Greenplum Database server configuration parameter `client_min_messages` to `LOG`. See the *Greenplum Database Reference Guide* for information about the parameter.

Examples

This example shows the differences for a query that is run against partitioned tables when GPORCA is enabled.

This `CREATE TABLE` statement creates a table with single level partitions:

```
CREATE TABLE sales (trans_id int, date date,
    amount decimal(9,2), region text)
   DISTRIBUTED BY (trans_id)
   PARTITION BY RANGE (date)
      (START (date '20160101')
       INCLUSIVE END (date '20170101')
       EXCLUSIVE EVERY (INTERVAL '1 month'),
   DEFAULT PARTITION outlying_dates )#
```

This query against the table is supported by GPORCA and does not generate errors in the log file:

```
select * from sales ;
```

The `EXPLAIN` plan output lists only the number of selected partitions.

```
 ->  Partition Selector for sales (dynamic scan id: 1)  (cost=10.00..100.00
 rows=50 width=4)
       Partitions selected:  13 (out of 13)
```

If a query against a partitioned table is not supported by GPORCA. Greenplum Database falls back to the Postgres Planner. The `EXPLAIN` plan generated by the Postgres Planner lists the selected partitions. This example shows a part of the explain plan that lists some selected partitions.

```
 ->  Append  (cost=0.00..0.00 rows=26 width=53)
     ->  Seq Scan on sales2_1_prt_7_2_prt_usa sales2  (cost=0.00..0.00
 rows=1 width=53)
     ->  Seq Scan on sales2_1_prt_7_2_prt_asia sales2  (cost=0.00..0.00
 rows=1 width=53)
     ...
```

This example shows the log output when the Greenplum Database falls back to the Postgres Planner from GPORCA.

When this query is run, Greenplum Database falls back to the Postgres Planner.

```
explain select * from pg_class;
```

A message is added to the log file. The message contains this `NOTICE` information that indicates the reason GPORCA did not execute the query:

```
NOTICE,""Feature not supported: Queries on master-only tables"
```

About Uniform Multi-level Partitioned Tables

GPORCA supports queries on a multi-level partitioned (MLP) table if the MLP table is a *uniform partitioned table*. A multi-level partitioned table is a partitioned table that was created with the `SUBPARTITION` clause. A uniform partitioned table must meet these requirements.

- The partitioned table structure is uniform. Each partition node at the same level must have the same hierarchical structure.
- The partition key constraints must be consistent and uniform. At each subpartition level, the sets of constraints on the child tables created for each branch must match.

You can display information about partitioned tables in several ways, including displaying information from these sources:

- The `pg_partitions` system view contains information on the structure of a partitioned table.
- The `pg_constraint` system catalog table contains information on table constraints.
- The psql meta command \d+ *tablename* displays the table constraints for child leaf tables of a partitioned table.

Example

This `CREATE TABLE` command creates a uniform partitioned table.

```
CREATE TABLE mlp (id  int,  year int,  month int,  day int,
   region  text)
   DISTRIBUTED  BY (id)
    PARTITION BY RANGE ( year)
      SUBPARTITION  BY LIST (region)
        SUBPARTITION TEMPLATE (
          SUBPARTITION usa  VALUES ( 'usa'),
          SUBPARTITION europe  VALUES ( 'europe'),
          SUBPARTITION asia  VALUES ( 'asia'))
   (  START ( 2006)  END ( 2016) EVERY ( 5));
```

These are child tables and the partition hierarchy that are created for the table `mlp`. This hierarchy consists of one subpartition level that contains two branches.

```
mlp_1_prt_11
   mlp_1_prt_11_2_prt_usa
   mlp_1_prt_11_2_prt_europe
   mlp_1_prt_11_2_prt_asia

mlp_1_prt_21
   mlp_1_prt_21_2_prt_usa
   mlp_1_prt_21_2_prt_europe
   mlp_1_prt_21_2_prt_asia
```

The hierarchy of the table is uniform, each partition contains a set of three child tables (subpartitions). The constraints for the region subpartitions are uniform, the set of constraints on the child tables for the branch table `mlp_1_prt_11` are the same as the constraints on the child tables for the branch table `mlp_1_prt_21`.

As a quick check, this query displays the constraints for the partitions.

```
WITH tbl AS (SELECT oid, partitionlevel AS level,
             partitiontablename AS part
          FROM pg_partitions, pg_class
          WHERE tablename = 'mlp' AND partitiontablename=relname
             AND partitionlevel=1 )
  SELECT tbl.part, consrc
    FROM tbl, pg_constraint
    WHERE tbl.oid = conrelid ORDER BY consrc;
```

Note: You will need modify the query for more complex partitioned tables. For example, the query does not account for table names in different schemas.

The `consrc` column displays constraints on the subpartitions. The set of region constraints for the subpartitions in `mlp_1_prt_1` match the constraints for the subpartitions in `mlp_1_prt_2`. The constraints for year are inherited from the parent branch tables.

```
           part           |               consrc
--------------------------+-------------------------------------
 mlp_1_prt_2_2_prt_asia   | (region = 'asia'::text)
 mlp_1_prt_1_2_prt_asia   | (region = 'asia'::text)
 mlp_1_prt_2_2_prt_europe | (region = 'europe'::text)
 mlp_1_prt_1_2_prt_europe | (region = 'europe'::text)
 mlp_1_prt_1_2_prt_usa    | (region = 'usa'::text)
 mlp_1_prt_2_2_prt_usa    | (region = 'usa'::text)
 mlp_1_prt_1_2_prt_asia   | ((year >= 2006) AND (year < 2011))
 mlp_1_prt_1_2_prt_usa    | ((year >= 2006) AND (year < 2011))
 mlp_1_prt_1_2_prt_europe | ((year >= 2006) AND (year < 2011))
 mlp_1_prt_2_2_prt_usa    | ((year >= 2011) AND (year < 2016))
 mlp_1_prt_2_2_prt_asia   | ((year >= 2011) AND (year < 2016))
 mlp_1_prt_2_2_prt_europe | ((year >= 2011) AND (year < 2016))
(12 rows)
```

If you add a default partition to the example partitioned table with this command:

```
ALTER TABLE mlp ADD DEFAULT PARTITION def
```

The partitioned table remains a uniform partitioned table. The branch created for default partition contains three child tables and the set of constraints on the child tables match the existing sets of child table constraints.

In the above example, if you drop the subpartition `mlp_1_prt_21_2_prt_asia` and add another subpartition for the region `canada`, the constraints are no longer uniform.

```
ALTER TABLE mlp ALTER PARTITION FOR (RANK(2))
  DROP PARTITION asia ;

ALTER TABLE mlp ALTER PARTITION FOR (RANK(2))
  ADD PARTITION canada VALUES ('canada');
```

Also, if you add a partition `canada` under `mlp_1_prt_21`, the partitioning hierarchy is not uniform.

However, if you add the subpartition `canada` to both `mlp_1_prt_21` and `mlp_1_prt_11` the of the original partitioned table, it remains a uniform partitioned table.

> **Note:** Only the constraints on the sets of partitions at a partition level must be the same. The names of the partitions can be different.

Defining Queries

Greenplum Database is based on the PostgreSQL implementation of the SQL standard.

This topic describes how to construct SQL queries in Greenplum Database.

- *SQL Lexicon*
- *SQL Value Expressions*

SQL Lexicon

SQL is a standard language for accessing databases. The language consists of elements that enable data storage, retrieval, analysis, viewing, manipulation, and so on. You use SQL commands to construct queries and commands that the Greenplum Database engine understands. SQL queries consist of a sequence of commands. Commands consist of a sequence of valid tokens in correct syntax order, terminated by a semicolon (`;`).

For more information about SQL commands, see *SQL Command Reference* in the *Greenplum Database Reference Guide*.

Greenplum Database uses PostgreSQL's structure and syntax, with some exceptions. For more information about SQL rules and concepts in PostgreSQL, see "SQL Syntax" in the PostgreSQL documentation.

SQL Value Expressions

SQL value expressions consist of one or more values, symbols, operators, SQL functions, and data. The expressions compare data or perform calculations and return a value as the result. Calculations include logical, arithmetic, and set operations.

The following are value expressions:

- An aggregate expression
- An array constructor
- A column reference
- A constant or literal value
- A correlated subquery
- A field selection expression
- A function call
- A new column value in an `INSERT` or `UPDATE`
- An operator invocation column reference
- A positional parameter reference, in the body of a function definition or prepared statement
- A row constructor
- A scalar subquery
- A search condition in a `WHERE` clause
- A target list of a `SELECT` command
- A type cast
- A value expression in parentheses, useful to group sub-expressions and override precedence
- A window expression

SQL constructs such as functions and operators are expressions but do not follow any general syntax rules. For more information about these constructs, see *Using Functions and Operators*.

Column References

A column reference has the form:

```
correlation.columnname
```

Here, `correlation` is the name of a table (possibly qualified with a schema name) or an alias for a table defined with a `FROM` clause or one of the keywords `NEW` or `OLD`. `NEW` and `OLD` can appear only in rewrite rules, but you can use other correlation names in any SQL statement. If the column name is unique across all tables in the query, you can omit the "`correlation.`" part of the column reference.

Positional Parameters

Positional parameters are arguments to SQL statements or functions that you reference by their positions in a series of arguments. For example, `$1` refers to the first argument, `$2` to the second argument, and so on. The values of positional parameters are set from arguments external to the SQL statement or supplied when SQL functions are invoked. Some client libraries support specifying data values separately from the

SQL command, in which case parameters refer to the out-of-line data values. A parameter reference has the form:

```
$number
```

For example:

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

Here, the `$1` references the value of the first function argument whenever the function is invoked.

Subscripts

If an expression yields a value of an array type, you can extract a specific element of the array value as follows:

```
expression[subscript]
```

You can extract multiple adjacent elements, called an array slice, as follows (including the brackets):

```
expression[lower_subscript:upper_subscript]
```

Each subscript is an expression and yields an integer value.

Array expressions usually must be in parentheses, but you can omit the parentheses when the expression to be subscripted is a column reference or positional parameter. You can concatenate multiple subscripts when the original array is multidimensional. For example (including the parentheses):

```
mytable.arraycolumn[4]
```

```
mytable.two_d_column[17][34]
```

```
$1[10:42]
```

```
(arrayfunction(a,b))[42]
```

Field Selection

If an expression yields a value of a composite type (row type), you can extract a specific field of the row as follows:

```
expression.fieldname
```

The row expression usually must be in parentheses, but you can omit these parentheses when the expression to be selected from is a table reference or positional parameter. For example:

```
mytable.mycolumn
```

```
$1.somecolumn
```

```
(rowfunction(a,b)).col3
```

A qualified column reference is a special case of field selection syntax.

Operator Invocations

Operator invocations have the following possible syntaxes:

```
expression operator expression(binary infix operator)
```

```
operator expression(unary prefix operator)
```

```
expression operator(unary postfix operator)
```

Where *operator* is an operator token, one of the key words `AND`, `OR`, or `NOT`, or qualified operator name in the form:

```
OPERATOR(schema.operatorname)
```

Available operators and whether they are unary or binary depends on the operators that the system or user defines. For more information about built-in operators, see *Built-in Functions and Operators*.

Function Calls

The syntax for a function call is the name of a function (possibly qualified with a schema name), followed by its argument list enclosed in parentheses:

```
function ([expression [, expression ... ]])
```

For example, the following function call computes the square root of 2:

```
sqrt(2)
```

See *Summary of Built-in Functions* in the *Greenplum Database Reference Guide* for lists of the built-in functions by category. You can add custom functions, too.

Aggregate Expressions

An aggregate expression applies an aggregate function across the rows that a query selects. An aggregate function performs a calculation on a set of values and returns a single value, such as the sum or average of the set of values. The syntax of an aggregate expression is one of the following:

- *aggregate_name*(*expression* [, ...]) [FILTER (WHERE *filter_clause*)] — operates across all input rows for which the expected result value is non-null. `ALL` is the default.
- *aggregate_name*(ALL *expression* [, ...]) [FILTER (WHERE *filter_clause*)] — operates identically to the first form because `ALL` is the default.
- *aggregate_name*(DISTINCT *expression* [, ...]) [FILTER (WHERE *filter_clause*)] — operates across all distinct non-null values of input rows.
- *aggregate_name*(*) [FILTER (WHERE *filter_clause*)] — operates on all rows with values both null and non-null. Generally, this form is most useful for the `count(*)` aggregate function.

Where *aggregate_name* is a previously defined aggregate (possibly schema-qualified) and *expression* is any value expression that does not contain an aggregate expression.

For example, `count(*)` yields the total number of input rows, `count(f1)` yields the number of input rows in which `f1` is non-null, and `count(distinct f1)` yields the number of distinct non-null values of `f1`.

If `FILTER` is specified, then only the input rows for which the *filter_clause* evaluates to true are fed to the aggregate function; other rows are discarded. For example:

```
SELECT
    count(*) AS unfiltered,
```

```
    count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
 unfiltered | filtered
------------+----------
         10 |        4
(1 row)
```

For predefined aggregate functions, see *Built-in Functions and Operators*. You can also add custom aggregate functions.

Greenplum Database provides the `MEDIAN` aggregate function, which returns the fiftieth percentile of the `PERCENTILE_CONT` result and special aggregate expressions for inverse distribution functions as follows:

```
PERCENTILE_CONT(_percentage_) WITHIN GROUP (ORDER BY _expression_)
```

```
PERCENTILE_DISC(_percentage_) WITHIN GROUP (ORDER BY _expression_)
```

Currently you can use only these two expressions with the keyword `WITHIN GROUP`.

Limitations of Aggregate Expressions

The following are current limitations of the aggregate expressions:

- Greenplum Database does not support the following keywords: `ALL`, `DISTINCT`, and `OVER`. See *Table 48: Advanced Aggregate Functions* for more details.
- An aggregate expression can appear only in the result list or `HAVING` clause of a `SELECT` command. It is forbidden in other clauses, such as `WHERE`, because those clauses are logically evaluated before the results of aggregates form. This restriction applies to the query level to which the aggregate belongs.
- When an aggregate expression appears in a subquery, the aggregate is normally evaluated over the rows of the subquery. If the aggregate's arguments (and *filter_clause* if any) contain only outer-level variables, the aggregate belongs to the nearest such outer level and evaluates over the rows of that query. The aggregate expression as a whole is then an outer reference for the subquery in which it appears, and the aggregate expression acts as a constant over any one evaluation of that subquery. The restriction about appearing only in the result list or `HAVING` clause applies with respect to the query level at which the aggregate appears. See *Scalar Subqueries* and *Table 46: Built-in functions and operators*.
- Greenplum Database does not support specifying an aggregate function as an argument to another aggregate function.
- Greenplum Database does not support specifying a window function as an argument to an aggregate function.

Window Expressions

Window expressions allow application developers to more easily compose complex online analytical processing (OLAP) queries using standard SQL commands. For example, with window expressions, users can calculate moving averages or sums over various intervals, reset aggregations and ranks as selected column values change, and express complex ratios in simple terms.

A window expression represents the application of a *window function* to a *window frame*, which is defined with an `OVER( )` clause. This is comparable to the type of calculation that can be done with an aggregate function and a `GROUP BY` clause. Unlike aggregate functions, which return a single result value for each group of rows, window functions return a result value for every row, but that value is calculated with respect to the set of rows in the window frame to which the row belongs. The `OVER( )` clause allows dividing the rows into *partitions* and then further restricting the window frame by specifying which rows preceding or following the current row within its partition to include in the calculation.

Greenplum Database does not support specifying a window function as an argument to another window function.

The syntax of a window expression is:

```
window_function ( [expression [, ...]] ) [ FILTER ( WHERE filter_clause ) ]
 OVER ( window_specification )
```

Where *window_function* is one of the functions listed in *Table 47: Window functions* or a user-defined window function, *expression* is any value expression that does not contain a window expression, and *window_specification* is:

```
[window_name]
[PARTITION BY expression [, ...]]
[[ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}]
 [, ...]
    [{RANGE | ROWS}
       { UNBOUNDED PRECEDING
       | expression PRECEDING
       | CURRENT ROW
       | BETWEEN window_frame_bound AND window_frame_bound }]]
```

and where *window_frame_bound* can be one of:

```
    UNBOUNDED PRECEDING
    expression PRECEDING
    CURRENT ROW
    expression FOLLOWING
    UNBOUNDED FOLLOWING
```

A window expression can appear only in the select list of a `SELECT` command. For example:

```
SELECT count(*) OVER(PARTITION BY customer_id), * FROM sales;
```

If `FILTER` is specified, then only the input rows for which the *filter_clause* evaluates to true are fed to the window function; other rows are discarded. In a window expression, a `FILTER` clause can be used only with a *window_function* that is an aggregate function.

In a window expression, the expression must contain an `OVER` clause. The `OVER` clause specifies the window frame—the rows to be processed by the window function. This syntactically distinguishes the function from a regular or aggregate function.

In a window aggregate function that is used in a window expression, Greenplum Database does not support a `DISTINCT` clause with multiple input expressions.

A window specification has the following characteristics:

- The `PARTITION BY` clause defines the window partitions to which the window function is applied. If omitted, the entire result set is treated as one partition.
- The `ORDER BY` clause defines the expression(s) for sorting rows within a window partition. The `ORDER BY` clause of a window specification is separate and distinct from the `ORDER BY` clause of a regular query expression. The `ORDER BY` clause is required for the window functions that calculate rankings, as it identifies the measure(s) for the ranking values. For OLAP aggregations, the `ORDER BY` clause is required to use window frames (the `ROWS` or `RANGE` clause).

  Note: Columns of data types without a coherent ordering, such as `time`, are not good candidates for use in the `ORDER BY` clause of a window specification. `Time`, with or without a specified time zone, lacks a coherent ordering because addition and subtraction do not have the expected effects. For example, the following is not generally true: `x::time < x::time + '2 hour'::interval`

- The `ROWS` or `RANGE` clause defines a window frame for aggregate (non-ranking) window functions. A window frame defines a set of rows within a window partition. When a window frame is defined, the window function computes on the contents of this moving frame rather than the fixed contents of the entire window partition. Window frames are row-based (`ROWS`) or value-based (`RANGE`).

Window Examples

The following examples demonstrate using window functions with partitions and window frames.

Example 1 – Aggregate Window Function Over a Partition

The `PARTITION BY` list in the `OVER` clause divides the rows into groups, or partitions, that have the same values as the specified expressions.

This example compares employees' salaries with the average salaries for their departments:

```
SELECT depname, empno, salary, avg(salary) OVER(PARTITION BY depname)
FROM empsalary;
  depname  | empno | salary |          avg
-----------+-------+--------+----------------------
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

The first three output columns come from the table `empsalary`, and there is one output row for each row in the table. The fourth column is the average calculated on all rows that have the same `depname` value as the current row. Rows that share the same `depname` value constitute a partition, and there are three partitions in this example. The `avg` function is the same as the regular `avg` aggregate function, but the `OVER` clause causes it to be applied as a window function.

You can also put the window specification in a `WINDOW` clause and reference it in the select list. This example is equivalent to the previous query:

```
SELECT depname, empno, salary, avg(salary) OVER(mywindow)
FROM empsalary
WINDOW mywindow AS (PARTITION BY depname);
```

Defining a named window is useful when the select list has multiple window functions using the same window specification.

Example 2 – Ranking Window Function With an ORDER BY Clause

An `ORDER BY` clause within the `OVER` clause controls the order in which rows are processed by window functions. The `ORDER BY` list for the window function does not have to match the output order of the query. This example uses the `rank()` window function to rank employees' salaries within their departments:

```
SELECT depname, empno, salary,
    rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
  depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    11 |   5200 |    2
 develop   |    10 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
```

```
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

Example 3 – Aggregate Function over a Row Window Frame

A `RANGE` or `ROWS` clause defines the window frame—a set of rows within a partition—that the window function includes in the calculation. `ROWS` specifies a physical set of rows to process, for example all rows from the beginning of the partition to the current row.

This example calculates a running total of employee's salaries by department using the `sum( )` function to total rows from the start of the partition to the current row:

```
SELECT depname, empno, salary,
    sum(salary) OVER (PARTITION BY depname ORDER BY salary
        ROWS between UNBOUNDED PRECEDING AND CURRENT ROW)
FROM empsalary ORDER BY depname, sum;
  depname  | empno | salary |  sum
-----------+-------+--------+-------
 develop   |     7 |   4200 |  4200
 develop   |     9 |   4500 |  8700
 develop   |    11 |   5200 | 13900
 develop   |    10 |   5200 | 19100
 develop   |     8 |   6000 | 25100
 personnel |     5 |   3500 |  3500
 personnel |     2 |   3900 |  7400
 sales     |     4 |   4800 |  4800
 sales     |     3 |   4800 |  9600
 sales     |     1 |   5000 | 14600
(10 rows)
```

Example 4 – Aggregate Function for a Range Window Frame

`RANGE` specifies logical values based on values of the `ORDER BY` expression in the `OVER` clause. This example demonstrates the difference between `ROWS` and `RANGE`. The frame contains all rows with salary values less than or equal to the current row. Unlike the previous example, for employees with the same salary, the sum is the same and includes the salaries of all of those employees.

```
SELECT depname, empno, salary,
    sum(salary) OVER (PARTITION BY depname ORDER BY salary
        RANGE between UNBOUNDED PRECEDING AND CURRENT ROW)
FROM empsalary ORDER BY depname, sum;
  depname  | empno | salary |  sum
-----------+-------+--------+-------
 develop   |     7 |   4200 |  4200
 develop   |     9 |   4500 |  8700
 develop   |    11 |   5200 | 19100
 develop   |    10 |   5200 | 19100
 develop   |     8 |   6000 | 25100
 personnel |     5 |   3500 |  3500
 personnel |     2 |   3900 |  7400
 sales     |     4 |   4800 |  9600
 sales     |     3 |   4800 |  9600
 sales     |     1 |   5000 | 14600
(10 rows)
```

Type Casts

A type cast specifies a conversion from one data type to another. A cast applied to a value expression of a known type is a run-time type conversion. The cast succeeds only if a suitable type conversion is defined. This differs from the use of casts with constants. A cast applied to a string literal represents the initial assignment of a type to a literal constant value, so it succeeds for any type if the contents of the string literal are acceptable input syntax for the data type.

Greenplum Database supports three types of casts applied to a value expression:

- *Explicit cast* - Greenplum Database applies a cast when you explicitly specify a cast between two data types. Greenplum Database accepts two equivalent syntaxes for explicit type casts:

  ```
  CAST ( expression AS type )
  expression::type
  ```

  The `CAST` syntax conforms to SQL; the syntax using `::` is historical PostgreSQL usage.
- *Assignment cast* - Greenplum Database implicitly invokes a cast in assignment contexts, when assigning a value to a column of the target data type. For example, a *`CREATE CAST`* command with the `AS ASSIGNMENT` clause creates a cast that is applied implicitly in the assignment context. This example assignment cast assumes that `tbl1.f1` is a column of type `text`. The `INSERT` command is allowed because the value is implicitly cast from the `integer` to `text` type.

  ```
  INSERT INTO tbl1 (f1) VALUES (42);
  ```
- *Implicit cast* - Greenplum Database implicitly invokes a cast in assignment or expression contexts. For example, a `CREATE CAST` command with the `AS IMPLICIT` clause creates an implicit cast, a cast that is applied implicitly in both the assignment and expression context. This example implicit cast assumes that `tbl1.c1` is a column of type `int`. For the calculation in the predicate, the value of `c1` is implicitly cast from `int` to a `decimal` type.

  ```
  SELECT * FROM tbl1 WHERE tbl1.c2 = (4.3 + tbl1.c1) ;
  ```

You can usually omit an explicit type cast if there is no ambiguity about the type a value expression must produce (for example, when it is assigned to a table column); the system automatically applies a type cast. Greenplum Database implicitly applies casts only to casts defined with a cast context of assignment or explicit in the system catalogs. Other casts must be invoked with explicit casting syntax to prevent unexpected conversions from being applied without the user's knowledge.

You can display cast information with the `psql` meta-command `\dC`. Cast information is stored in the catalog table `pg_cast`, and type information is stored in the catalog table `pg_type`.

Scalar Subqueries

A scalar subquery is a `SELECT` query in parentheses that returns exactly one row with one column. Do not use a `SELECT` query that returns multiple rows or columns as a scalar subquery. The query runs and uses the returned value in the surrounding value expression. A correlated scalar subquery contains references to the outer query block.

Correlated Subqueries

A correlated subquery (CSQ) is a `SELECT` query with a `WHERE` clause or target list that contains references to the parent outer clause. CSQs efficiently express results in terms of results of another query. Greenplum Database supports correlated subqueries that provide compatibility with many existing applications. A CSQ is a scalar or table subquery, depending on whether it returns one or multiple rows. Greenplum Database does not support correlated subqueries with skip-level correlations.

Correlated Subquery Examples

Example 1 – Scalar correlated subquery

```
SELECT * FROM t1 WHERE t1.x
           > (SELECT MAX(t2.x) FROM t2 WHERE t2.y = t1.y);
```

Example 2 – Correlated EXISTS subquery

```
SELECT * FROM t1 WHERE
EXISTS (SELECT 1 FROM t2 WHERE t2.x = t1.x);
```

Greenplum Database uses one of the following methods to run CSQs:

- Unnest the CSQ into join operations – This method is most efficient, and it is how Greenplum Database runs most CSQs, including queries from the TPC-H benchmark.
- Run the CSQ on every row of the outer query – This method is relatively inefficient, and it is how Greenplum Database runs queries that contain CSQs in the `SELECT` list or are connected by `OR` conditions.

The following examples illustrate how to rewrite some of these types of queries to improve performance.

Example 3 - CSQ in the Select List

Original Query

```
SELECT T1.a,
      (SELECT COUNT(DISTINCT T2.z) FROM t2 WHERE t1.x = t2.y) dt2
FROM t1;
```

Rewrite this query to perform an inner join with `t1` first and then perform a left join with `t1` again. The rewrite applies for only an equijoin in the correlated condition.

Rewritten Query

```
SELECT t1.a, dt2 FROM t1
       LEFT JOIN
        (SELECT t2.y AS csq_y, COUNT(DISTINCT t2.z) AS dt2
              FROM t1, t2 WHERE t1.x = t2.y
              GROUP BY t1.x)
       ON (t1.x = csq_y);
```

Example 4 - CSQs connected by OR Clauses

Original Query

```
SELECT * FROM t1
WHERE
x > (SELECT COUNT(*) FROM t2 WHERE t1.x = t2.x)
OR x < (SELECT COUNT(*) FROM t3 WHERE t1.y = t3.y)
```

Rewrite this query to separate it into two parts with a union on the `OR` conditions.

Rewritten Query

```
SELECT * FROM t1
WHERE x > (SELECT count(*) FROM t2 WHERE t1.x = t2.x)
UNION
SELECT * FROM t1
```

```
WHERE x < (SELECT count(*) FROM t3 WHERE t1.y = t3.y)
```

To view the query plan, use `EXPLAIN SELECT` or `EXPLAIN ANALYZE SELECT`. Subplan nodes in the query plan indicate that the query will run on every row of the outer query, and the query is a candidate for rewriting. For more information about these statements, see *Query Profiling*.

Array Constructors

An array constructor is an expression that builds an array value from values for its member elements. A simple array constructor consists of the key word `ARRAY`, a left square bracket `[`, one or more expressions separated by commas for the array element values, and a right square bracket `]`. For example,

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
```

The array element type is the common type of its member expressions, determined using the same rules as for `UNION` or `CASE` constructs.

You can build multidimensional array values by nesting array constructors. In the inner constructors, you can omit the keyword `ARRAY`. For example, the following two `SELECT` statements produce the same result:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
```

Since multidimensional arrays must be rectangular, inner constructors at the same level must produce sub-arrays of identical dimensions.

Multidimensional array constructor elements are not limited to a sub-`ARRAY` construct; they are anything that produces an array of the proper kind. For example:

```
CREATE TABLE arr(f1 int[], f2 int[]);
INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]],
ARRAY[[5,6],[7,8]]);
SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
------------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
```

You can construct an array from the results of a subquery. Write the array constructor with the keyword `ARRAY` followed by a subquery in parentheses. For example:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                           ?column?
------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31}
```

The subquery must return a single column. The resulting one-dimensional array has an element for each row in the subquery result, with an element type matching that of the subquery's output column. The subscripts of an array value built with `ARRAY` always begin with `1`.

Row Constructors

A row constructor is an expression that builds a row value (also called a composite value) from values for its member fields. For example,

```
SELECT ROW(1,2.5,'this is a test');
```

Row constructors have the syntax `rowvalue.*`, which expands to a list of the elements of the row value, as when you use the syntax `.*` at the top level of a `SELECT` list. For example, if table `t` has columns `f1` and `f2`, the following queries are the same:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

By default, the value created by a `ROW` expression has an anonymous record type. If necessary, it can be cast to a named composite type — either the row type of a table, or a composite type created with `CREATE TYPE AS`. To avoid ambiguity, you can explicitly cast the value if necessary. For example:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);
CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1'
LANGUAGE SQL;
```

In the following query, you do not need to cast the value because there is only one `getf1()` function and therefore no ambiguity:

```
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);
CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT
$1.f1' LANGUAGE SQL;
```

Now we need a cast to indicate which function to call:

```
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique
```

```
SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS
myrowtype));
 getf1
-------
    11
```

You can use row constructors to build composite values to be stored in a composite-type table column or to be passed to a function that accepts a composite parameter.

Expression Evaluation Rules

The order of evaluation of subexpressions is undefined. The inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order.

If you can determine the result of an expression by evaluating only some parts of the expression, then other subexpressions might not be evaluated at all. For example, in the following expression:

```
SELECT true OR somefunc();
```

`somefunc()` would probably not be called at all. The same is true in the following expression:

```
SELECT somefunc() OR true;
```

This is not the same as the left-to-right evaluation order that Boolean operators enforce in some programming languages.

Do not use functions with side effects as part of complex expressions, especially in `WHERE` and `HAVING` clauses, because those clauses are extensively reprocessed when developing an execution plan. Boolean expressions (`AND`/`OR`/`NOT` combinations) in those clauses can be reorganized in any manner that Boolean algebra laws allow.

Use a `CASE` construct to force evaluation order. The following example is an untrustworthy way to avoid division by zero in a `WHERE` clause:

```
SELECT ... WHERE x <> 0 AND y/x > 1.5;
```

The following example shows a trustworthy evaluation order:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false
END;
```

This `CASE` construct usage defeats optimization attempts; use it only when necessary.

WITH Queries (Common Table Expressions)

The `WITH` clause provides a way to use subqueries or perform a data modifying operation in a larger `SELECT` query. You can also use the `WITH` clause in an `INSERT`, `UPDATE`, or `DELETE` command.

See *SELECT in a WITH Clause* for information about using `SELECT` in a `WITH` clause.

See *Data-Modifying Statements in a WITH clause*, for information about using `INSERT`, `UPDATE`, or `DELETE` in a `WITH` clause.

Note: These are limitations for using a `WITH` clause.

- For a `SELECT` command that includes a `WITH` clause, the clause can contain at most a single clause that modifies table data (`INSERT`, `UPDATE`, or `DELETE` command).
- For a data-modifying command (`INSERT`, `UPDATE`, or `DELETE`) that includes a `WITH` clause, the clause can only contain a `SELECT` command, the `WITH` clause cannot contain a data-modifying command.

By default, the `RECURSIVE` keyword for the `WITH` clause is enabled. `RECURSIVE` can be disabled by setting the server configuration parameter *`gp_recursive_cte`* to `false`.

SELECT in a WITH Clause

The subqueries, which are often referred to as Common Table Expressions or CTEs, can be thought of as defining temporary tables that exist just for the query. These examples show the `WITH` clause being used with a `SELECT` command. The example `WITH` clauses can be used the same way with `INSERT`, `UPDATE`, or `DELETE`. In each case, the `WITH` clause effectively provides temporary tables that can be referred to in the main command.

A `SELECT` command in the `WITH` clause is evaluated only once per execution of the parent query, even if it is referred to more than once by the parent query or sibling `WITH` clauses. Thus, expensive calculations that are needed in multiple places can be placed within a `WITH` clause to avoid redundant work. Another

possible application is to prevent unwanted multiple evaluations of functions with side-effects. However, the other side of this coin is that the optimizer is less able to push restrictions from the parent query down into a WITH query than an ordinary sub-query. The WITH query will generally be evaluated as written, without suppression of rows that the parent query might discard afterwards. However, evaluation might stop early if the references to the query demand only a limited number of rows.

One use of this feature is to break down complicated queries into simpler parts. This example query displays per-product sales totals in only the top sales regions:

```
WITH regional_sales AS (
     SELECT region, SUM(amount) AS total_sales
     FROM orders
     GROUP BY region
  ), top_regions AS (
     SELECT region
     FROM regional_sales
     WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
  )
SELECT region,
    product,
    SUM(quantity) AS product_units,
    SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

The query could have been written without the WITH clause, but would have required two levels of nested sub-SELECTs. It is easier to follow with the WITH clause.

When the optional RECURSIVE keyword is enabled, the WITH clause can accomplish things not otherwise possible in standard SQL. Using RECURSIVE, a query in the WITH clause can refer to its own output. This is a simple example that computes the sum of integers from 1 through 100:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

The general form of a recursive WITH clause (a WITH clause that uses a the RECURSIVE keyword) is a *non-recursive term*, followed by a UNION (or UNION ALL), and then a *recursive term*, where only the *recursive term* can contain a reference to the query output.

```
non_recursive_term UNION [ ALL ] recursive_term
```

A recursive WITH query that contains a UNION [ALL] is executed as follows:

1. Evaluate the non-recursive term. For UNION (but not UNION ALL), discard duplicate rows. Include all remaining rows in the result of the recursive query, and also place them in a temporary *working table*.
2. As long as the working table is not empty, repeat these steps:
    a. Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For UNION (but not UNION ALL), discard duplicate rows and rows that duplicate any previous result row. Include all remaining rows in the result of the recursive query, and also place them in a temporary *intermediate table*.
    b. Replace the contents of the *working table* with the contents of the *intermediate table*, then empty the *intermediate table*.

Note: Strictly speaking, the process is iteration not recursion, but RECURSIVE is the terminology chosen by the SQL standards committee.

Recursive `WITH` queries are typically used to deal with hierarchical or tree-structured data. An example is this query to find all the direct and indirect sub-parts of a product, given only a table that shows immediate inclusions:

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part ;
```

When working with recursive `WITH` queries, you must ensure that the recursive part of the query eventually returns no tuples, or else the query loops indefinitely. In the example that computes the sum of integers, the working table contains a single row in each step, and it takes on the values from 1 through 100 in successive steps. In the 100th step, there is no output because of the `WHERE` clause, and the query terminates.

For some queries, using `UNION` instead of `UNION ALL` can ensure that the recursive part of the query eventually returns no tuples by discarding rows that duplicate previous output rows. However, often a cycle does not involve output rows that are complete duplicates: it might be sufficient to check just one or a few fields to see if the same point has been reached before. The standard method for handling such situations is to compute an array of the visited values. For example, consider the following query that searches a table graph using a link field:

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
        SELECT g.id, g.link, g.data, 1
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1
        FROM graph g, search_graph sg
        WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

This query loops if the link relationships contain cycles. Because the query requires a `depth` output, changing `UNION ALL` to `UNION` does not eliminate the looping. Instead the query needs to recognize whether it has reached the same row again while following a particular path of links. This modified query adds two columns, `path` and `cycle`, to the loop-prone query:

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1,
          ARRAY[g.id],
          false
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1,
          path || g.id,
          g.id = ANY(path)
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

Aside from detecting cycles, the array value of `path` is useful in its own right since it represents the path taken to reach any particular row.

In the general case where more than one field needs to be checked to recognize a cycle, an array of rows can be used. For example, if we needed to compare fields `f1` and `f2`:

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1,
          ARRAY[ROW(g.f1, g.f2)],
          false
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1,
          path || ROW(g.f1, g.f2),
          ROW(g.f1, g.f2) = ANY(path)
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

Tip: Omit the `ROW()` syntax in the case where only one field needs to be checked to recognize a cycle. This uses a simple array rather than a composite-type array, gaining efficiency.

Tip: The recursive query evaluation algorithm produces its output in breadth-first search order. You can display the results in depth-first search order by making the outer query `ORDER BY` a path column constructed in this way.

A helpful technique for testing a query when you are not certain if it might loop indefinitely is to place a `LIMIT` in the parent query. For example, this query would loop forever without the `LIMIT` clause:

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

The technique works because the recursive `WITH` implementation evaluates only as many rows of a `WITH` query as are actually fetched by the parent query. Using this technique in production is not recommended, because other systems might work differently. Also, the technique might not work if the outer query sorts the recursive `WITH` results or join the results to another table.

Data-Modifying Statements in a WITH clause

For a `SELECT` command, you can use the data-modifying commands `INSERT`, `UPDATE`, or `DELETE` in a `WITH` clause. This allows you to perform several different operations in the same query.

A data-modifying statement in a `WITH` clause is executed exactly once, and always to completion, independently of whether the primary query reads all (or indeed any) of the output. This is different from the rule when using `SELECT` in a `WITH` clause, the execution of a `SELECT` continues only as long as the primary query demands its output.

This simple CTE query deletes rows from `products`. The `DELETE` in the `WITH` clause deletes the specified rows from products, returning their contents by means of its `RETURNING` clause.

```
WITH deleted_rows AS (
    DELETE FROM products
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
SELECT * FROM deleted_rows;
```

Data-modifying statements in a WITH clause must have RETURNING clauses, as shown in the previous example. It is the output of the RETURNING clause, not the target table of the data-modifying statement, that forms the temporary table that can be referred to by the rest of the query. If a data-modifying statement in a WITH lacks a RETURNING clause, an error is returned.

If the optional RECURSIVE keyword is enabled, recursive self-references in data-modifying statements are not allowed. In some cases it is possible to work around this limitation by referring to the output of a recursive WITH. For example, this query would remove all direct and indirect subparts of a product.

```
WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
DELETE FROM parts
  WHERE part IN (SELECT part FROM included_parts);
```

The sub-statements in a WITH clause are executed concurrently with each other and with the main query. Therefore, when using a data-modifying statement in a WITH, the statement is executed in a *snapshot*. The effects of the statement are not visible on the target tables. The RETURNING data is the only way to communicate changes between different WITH sub-statements and the main query. In this example, the outer SELECT returns the original prices before the action of the UPDATE in the WITH clause.

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM products;
```

In this example the outer SELECT returns the updated data.

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

Updating the same row twice in a single statement is not supported. The effects of such a statement will not be predictable. Only one of the modifications takes place, but it is not easy (and sometimes not possible) to predict which modification occurs.

Any table used as the target of a data-modifying statement in a WITH clause must not have a conditional rule, or an ALSO rule, or an INSTEAD rule that expands to multiple statements.

Using Functions and Operators

Description of user-defined and built-in functions and operators in Greenplum Database.

- *Using Functions in Greenplum Database*
- *User-Defined Functions*
- *Built-in Functions and Operators*
- *Window Functions*
- *Advanced Aggregate Functions*

Using Functions in Greenplum Database

When you invoke a function in Greenplum Database, function attributes control the execution of the function. The volatility attributes (IMMUTABLE, STABLE, VOLATILE) and the EXECUTE ON attributes

control two different aspects of function execution. In general, volatility indicates when the function is executed, and `EXECUTE ON` indicates where it is executed. The volatility attributes are PostgreSQL based attributes, the `EXECUTE ON` attributes are Greenplum Database attributes.

For example, a function defined with the `IMMUTABLE` attribute can be executed at query planning time, while a function with the `VOLATILE` attribute must be executed for every row in the query. A function with the `EXECUTE ON MASTER` attribute executes only on the master instance, and a function with the `EXECUTE ON ALL SEGMENTS` attribute executes on all primary segment instances (not the master).

These tables summarize what Greenplum Database assumes about function execution based on the attribute.

Table 44: Function Volatility Attributes in Greenplum Database

Function Attribute	Greenplum Support	Description	Comments
IMMUTABLE	Yes	Relies only on information directly in its argument list. Given the same argument values, always returns the same result.	
STABLE	Yes, in most cases	Within a single table scan, returns the same result for same argument values, but results change across SQL statements.	Results depend on database lookups or parameter values. `current_timestamp` family of functions is `STABLE`; values do not change within an execution.
VOLATILE	Restricted	Function values can change within a single table scan. For example: `random()`, `timeofday()`. This is the default attribute.	Any function with side effects is volatile, even if its result is predictable. For example: `setval()`.

Table 45: Function EXECUTE ON attributes in Greenplum Database

Function Attribute	Description	Comments
EXECUTE ON ANY	Indicates that the function can be executed on the master, or any segment instance, and it returns the same result regardless of where it executes. This is the default attribute.	Greenplum Database determines where the function executes.
EXECUTE ON MASTER	Indicates that the function must be executed on the master instance.	Specify this attribute if the user-defined function executes queries to access tables.
EXECUTE ON ALL SEGMENTS	Indicates that for each invocation, the function must be executed on all primary segment instances, but not the master.	
EXECUTE ON INITPLAN	Indicates that the function contains an SQL command that dispatches queries to the segment instances and requires special processing on the master instance by Greenplum Database when possible.	

You can display the function volatility and `EXECUTE ON` attribute information with the psql `\df+` *`function`* command.

Refer to the PostgreSQL *Function Volatility Categories* documentation for additional information about the Greenplum Database function volatility classifications.

For more information about `EXECUTE ON` attributes, see *`CREATE FUNCTION`*.

In Greenplum Database, data is divided up across segments — each segment is a distinct PostgreSQL database. To prevent inconsistent or unexpected results, do not execute functions classified as `VOLATILE` at the segment level if they contain SQL commands or modify the database in any way. For example, functions such as `setval()` are not allowed to execute on distributed data in Greenplum Database because they can cause inconsistent data between segment instances.

A function can execute read-only queries on replicated tables (`DISTRIBUTED REPLICATED`) on the segments, but any SQL command that modifies data must execute on the master instance.

> **Note:** The hidden system columns (`ctid`, `cmin`, `cmax`, `xmin`, `xmax`, and `gp_segment_id`) cannot be referenced in user queries on replicated tables because they have no single, unambiguous value. Greenplum Database returns a `column does not exist` error for the query.

To ensure data consistency, you can safely use `VOLATILE` and `STABLE` functions in statements that are evaluated on and run from the master. For example, the following statements run on the master (statements without a `FROM` clause):

```
SELECT setval('myseq', 201);
SELECT foo();
```

If a statement has a `FROM` clause containing a distributed table *and* the function in the `FROM` clause returns a set of rows, the statement can run on the segments:

```
SELECT * from foo();
```

Greenplum Database does not support functions that return a table reference (`rangeFuncs`) or functions that use the `refCursor` data type.

Function Volatility and Plan Caching

There is relatively little difference between the `STABLE` and `IMMUTABLE` function volatility categories for simple interactive queries that are planned and immediately executed. It does not matter much whether a function is executed once during planning or once during query execution start up. But there is a big difference when you save the plan and reuse it later. If you mislabel a function `IMMUTABLE`, Greenplum Database may prematurely fold it to a constant during planning, possibly reusing a stale value during subsequent execution of the plan. You may run into this hazard when using `PREPARE`d statements, or when using languages such as PL/pgSQL that cache plans.

User-Defined Functions

Greenplum Database supports user-defined functions. See *Extending SQL* in the PostgreSQL documentation for more information.

Use the `CREATE FUNCTION` statement to register user-defined functions that are used as described in *Using Functions in Greenplum Database*. By default, user-defined functions are declared as `VOLATILE`, so if your user-defined function is `IMMUTABLE` or `STABLE`, you must specify the correct volatility level when you register your function.

By default, user-defined functions are declared as `EXECUTE ON ANY`. A function that executes queries to access tables is supported only when the function executes on the master instance, except that a function can execute `SELECT` commands that access only replicated tables on the segment instances. A function that accesses hash-distributed or randomly distributed tables must be defined with the `EXECUTE ON`

MASTER attribute. Otherwise, the function might return incorrect results when the function is used in a complicated query. Without the attribute, planner optimization might determine it would be beneficial to push the function invocation to segment instances.

When you create user-defined functions, avoid using fatal errors or destructive calls. Greenplum Database may respond to such errors with a sudden shutdown or restart.

In Greenplum Database, the shared library files for user-created functions must reside in the same library path location on every host in the Greenplum Database array (masters, segments, and mirrors).

You can also create and execute anonymous code blocks that are written in a Greenplum Database procedural language such as PL/pgSQL. The anonymous blocks run as transient anonymous functions. For information about creating and executing anonymous blocks, see the *DO* command.

Built-in Functions and Operators

The following table lists the categories of built-in functions and operators supported by PostgreSQL. All functions and operators are supported in Greenplum Database as in PostgreSQL with the exception of STABLE and VOLATILE functions, which are subject to the restrictions noted in *Using Functions in Greenplum Database*. See the *Functions and Operators* section of the PostgreSQL documentation for more information about these built-in functions and operators.

Greenplum Database includes JSON processing functions that manipulate values the `json` data type. For information about JSON data, see *Working with JSON Data*.

Table 46: Built-in functions and operators

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
Logical Operators			
Comparison Operators			
Mathematical Functions and Operators	random setseed		
String Functions and Operators	*All built-in conversion functions*	convert pg_client_encoding	
Binary String Functions and Operators			
Bit String Functions and Operators			
Pattern Matching			
Data Type Formatting Functions		to_char to_timestamp	

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
Date/Time Functions and Operators	timeofday	age current_date current_time current_timestamp localtime localtimestamp now	
Enum Support Functions			
Geometric Functions and Operators			
Network Address Functions and Operators			
Sequence Manipulation Functions	nextval() setval()		
Conditional Expressions			
Array Functions and Operators		*All array functions*	
Aggregate Functions			
Subquery Expressions			
Row and Array Comparisons			
Set Returning Functions	generate_series		
System Information Functions		*All session information functions* *All access privilege inquiry functions* *All schema visibility inquiry functions* *All system catalog information functions* *All comment information functions* *All transaction ids and snapshots*	

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
System Administration Functions	set_config pg_cancel_backend pg_terminate_backend pg_reload_conf pg_rotate_logfile pg_start_backup pg_stop_backup pg_size_pretty pg_ls_dir pg_read_file pg_stat_file	current_setting *All database object size functions*	**Note:** The function `pg_column_size` displays bytes required to store the value, possibly with TOAST compression.

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
XML Functions and function-like expressions		cursor_to_xml(cursor refcursor, count int, nulls boolean, tableforest boolean, targetns text) cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text) database_to_xml(nulls boolean, tableforest boolean, targetns text) database_to_xmlschema(nulls boolean, tableforest boolean, targetns text) database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text) query_to_xml(query text, nulls boolean, tableforest boolean, targetns text) query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text) query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text) schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text) schema_to_xmlschema(schema name, nulls boolean,	

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
		tableforest boolean, targetns text) schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text) table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text) table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text) table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text) xmlagg(xml) xmlconcat(xml[, ...]) xmlelement(name name [, xmlattributes(value [AS attname] [, ...])] [, content, ...]) xmlexists(text, xml) xmlforest(content [AS name] [, ...]) xml_is_well_formed(text) xml_is_well_formed_document(text) xml_is_well_formed_content(text) xmlparse ({ DOCUMENT | CONTENT } value) xpath(text, xml) xpath(text, xml, text[])	

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
		xpath_exists(text, xml) xpath_exists(text, xml, text[]) xmlpi(name target [, content]) xmlroot(xml, version text | no value [, standalone yes|no|no value]) xmlserialize ({ DOCUMENT | CONTENT } value AS type) xml(text) text(xml) xmlcomment(xml) xmlconcat2(xml, xml)	

Window Functions

The following built-in window functions are Greenplum extensions to the PostgreSQL database. All window functions are *immutable*. For more information about window functions, see *Window Expressions*.

Table 47: Window functions

Function	Return Type	Full Syntax	Description
`cume_dist()`	`double precision`	`CUME_DIST() OVER ( [PARTITION BY expr ] ORDER BY expr )`	Calculates the cumulative distribution of a value in a group of values. Rows with equal values always evaluate to the same cumulative distribution value.
`dense_rank()`	`bigint`	`DENSE_RANK () OVER ( [PARTITION BY expr ] ORDER BY expr )`	Computes the rank of a row in an ordered group of rows without skipping rank values. Rows with equal values are given the same rank value.
`first_value(`*`expr`*`)`	same as input expr type	`FIRST_VALUE( expr ) OVER ( [PARTITION BY expr ] ORDER BY expr [ROWS|RANGE frame_expr ] )`	Returns the first value in an ordered set of values.

Function	Return Type	Full Syntax	Description
lag(*expr* [,*offset*] [,*default*])	same as input *expr* type	LAG(*expr* [, *offset*] [, *default*]) OVER ([PARTITION BY *expr*] ORDER BY *expr*)	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, LAG provides access to a row at a given physical offset prior to that position. The default offset is 1. *default* sets the value that is returned if the offset goes beyond the scope of the window. If *default* is not specified, the default value is null.
last_value(*expr*)	same as input *expr* type	LAST_VALUE(*expr*) OVER ([PARTITION BY *expr*] ORDER BY *expr* [ROWS|RANGE *frame_expr*])	Returns the last value in an ordered set of values.
lead(*expr* [,*offset*] [,*default*])	same as input *expr* type	LEAD(*expr* [,*offset*] [,*exprdefault*]) OVER ([PARTITION BY *expr*] ORDER BY *expr*)	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, lead provides access to a row at a given physical offset after that position. If *offset* is not specified, the default offset is 1. *default* sets the value that is returned if the offset goes beyond the scope of the window. If *default* is not specified, the default value is null.
ntile(*expr*)	bigint	NTILE(*expr*) OVER ([PARTITION BY *expr*] ORDER BY *expr*)	Divides an ordered data set into a number of buckets (as defined by *expr*) and assigns a bucket number to each row.

Function	Return Type	Full Syntax	Description
`percent_rank()`	`double precision`	`PERCENT_RANK () OVER ( [PARTITION BY expr] ORDER BY expr )`	Calculates the rank of a hypothetical row R minus 1, divided by 1 less than the number of rows being evaluated (within a window partition).
`rank()`	`bigint`	`RANK () OVER ( [PARTITION BY expr] ORDER BY expr )`	Calculates the rank of a row in an ordered group of values. Rows with equal values for the ranking criteria receive the same rank. The number of tied rows are added to the rank number to calculate the next rank value. Ranks may not be consecutive numbers in this case.
`row_number()`	`bigint`	`ROW_NUMBER () OVER ( [PARTITION BY expr] ORDER BY expr )`	Assigns a unique number to each row to which it is applied (either each row in a window partition or each row of the query).

Advanced Aggregate Functions

The following built-in advanced aggregate functions are Greenplum extensions of the PostgreSQL database. These functions are *immutable*.

> **Note:** The Greenplum MADlib Extension for Analytics provides additional advanced functions to perform statistical analysis and machine learning with Greenplum Database data. See *Greenplum MADlib Extension for Analytics* in the *Greenplum Database Reference Guide*.

Table 48: Advanced Aggregate Functions

Function	Return Type	Full Syntax	Description

`MEDIAN (expr)`	`timestamp, timestamptz, interval, float`	`MEDIAN (expression)`
		Example:
		`SELECT departmzent_id, MEDIAN(salary) FROM employees GROUP BY department_id;`
			Can take a two-dimensional array as input. Treats such arrays as matrices.

Function	Return Type	Full Syntax	Description

`sum(array[])`	`smallint[], int[], bigint[], float[]`	`sum(array[[1,2],[3,4]])`
		Example:
		`CREATE TABLE mymatrix (myvalue int[]);`
		`INSERT INTO mymatrix VALUES (array[[1,2], [3,4]]);`
		`INSERT INTO mymatrix VALUES (array[[0,1], [1,0]]);`
		`SELECT sum(myvalue) FROM mymatrix;`
		`sum`
		`---------------`
		`{{1,3},{4,4}}`
			Performs matrix summation. Can take as input a two-dimensional array that is treated as a matrix.

`pivot_sum (label[], label, expr)`	`int[], bigint[], float[]`	`pivot_ sum( array['A1','A2'], attr, value)`	A pivot aggregation using sum to resolve duplicate entries.
`unnest (array[])`	set of `anyelement`	`unnest( array['one', 'row', 'per', 'item'])`	Transforms a one dimensional array into rows. Returns a set of `anyelement`, a polymorphic *pseudo-type* in PostgreSQL.

Working with JSON Data

Greenplum Database supports the `json` and `jsonb` data types that store JSON (JavaScript Object Notation) data.

Greenplum Database supports JSON as specified in the *RFC 7159* document and enforces data validity according to the JSON rules. There are also JSON-specific functions and operators available for the `json` and `jsonb` data types. See *JSON Functions and Operators*.

This section contains the following topics:

- *About JSON Data*
- *JSON Input and Output Syntax*
- *Designing JSON documents*
- *jsonb Containment and Existence*
- *jsonb Indexing*
- *JSON Functions and Operators*

About JSON Data

Greenplum Database supports two JSON data types: `json` and `jsonb`. They accept almost identical sets of values as input. The major difference is one of efficiency.

- The `json` data type stores an exact copy of the input text. This requires JSON processing functions to reparse `json` data on each execution. The `json` data type does not alter the input text.

    - Semantically-insignificant white space between tokens is retained, as well as the order of keys within JSON objects.
    - All key/value pairs are kept even if a JSON object contains duplicate keys. For duplicate keys, JSON processing functions consider the last value as the operative one.
- The `jsonb` data type stores a decomposed binary format of the input text. The conversion overhead makes data input slightly slower than the `json` data type. However, The JSON processing functions are significantly faster because reparsing `jsonb` data is not required. The `jsonb` data type alters the input text.

    - White space is not preserved.
    - The order of object keys is not preserved.
    - Duplicate object keys are not kept. If the input includes duplicate keys, only the last value is kept.

    The `jsonb` data type supports indexing. See *jsonb Indexing*.

In general, JSON data should be stored as the `jsonb` data type unless there are specialized needs, such as legacy assumptions about ordering of object keys.

About Unicode Characters in JSON Data

The *RFC 7159* document permits JSON strings to contain Unicode escape sequences denoted by `\u`*XXXX*. However, Greenplum Database allows only one character set encoding per database. It is not possible for the `json` data type to conform rigidly to the JSON specification unless the database encoding is UTF8. Attempts to include characters that cannot be represented in the database encoding will fail. Characters that can be represented in the database encoding, but not in UTF8, are allowed.

- The Greenplum Database input function for the `json` data type allows Unicode escapes regardless of the database encoding and checks Unicode escapes only for syntactic correctness (a `\u` followed by four hex digits).
- The Greenplum Database input function for the `jsonb` data type is more strict. It does not allow Unicode escapes for non-ASCII characters (those above `U+007F`) unless the database encoding is UTF8. It also rejects `\u0000`, which cannot be represented in the Greenplum Database `text` type, and it requires that any use of Unicode surrogate pairs to designate characters outside the Unicode Basic Multilingual Plane be correct. Valid Unicode escapes, except for `\u0000`, are converted to the equivalent ASCII or UTF8 character for storage; this includes folding surrogate pairs into a single character.

    Note: Many of the JSON processing functions described in *JSON Functions and Operators* convert Unicode escapes to regular characters. The functions throw an error for characters that cannot be represented in the database encoding. You should avoid mixing Unicode escapes in JSON with a non-UTF8 database encoding, if possible.

Mapping JSON Data Types to Greenplum Data Types

When converting JSON text input into `jsonb` data, the primitive data types described by RFC 7159 are effectively mapped onto native Greenplum Database data types, as shown in the following table.

Table 49: JSON Primitive Types and Corresponding Greenplum Database Data Types

JSON primitive data type	Greenplum Database data type	Notes
`string`	`text`	`\u0000` is not allowed. Non-ASCII Unicode escapes are allowed only if database encoding is UTF8
`number`	`numeric`	`NaN` and `infinity` values are disallowed

JSON primitive data type	Greenplum Database data type	Notes
`boolean`	`boolean`	Only lowercase `true` and `false` spellings are accepted
`null`	(none)	The JSON `null` primitive type is different than the SQL `NULL`.

There are some minor constraints on what constitutes valid `jsonb` data that do not apply to the `json` data type, nor to JSON in the abstract, corresponding to limits on what can be represented by the underlying data type. Notably, when converting data to the `jsonb` data type, numbers that are outside the range of the Greenplum Database `numeric` data type are rejected, while the `json` data type does not reject such numbers.

Such implementation-defined restrictions are permitted by RFC 7159. However, in practice such problems might occur in other implementations, as it is common to represent the JSON `number` primitive type as IEEE 754 double precision floating point (which RFC 7159 explicitly anticipates and allows for).

When using JSON as an interchange format with other systems, be aware of the possibility of losing numeric precision compared to data originally stored by Greenplum Database.

Also, as noted in the previous table, there are some minor restrictions on the input format of JSON primitive types that do not apply to the corresponding Greenplum Database data types.

JSON Input and Output Syntax

The input and output syntax for the `json` data type is as specified in RFC 7159.

The following are all valid `json` expressions:

```
-- Simple scalar/primitive value
-- Primitive values can be numbers, quoted strings, true, false, or null
SELECT '5'::json;

-- Array of zero or more elements (elements need not be of same type)
SELECT '[1, 2, "foo", null]'::json;

-- Object containing pairs of keys and values
-- Note that object keys must always be quoted strings
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- Arrays and objects can be nested arbitrarily
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

As previously stated, when a JSON value is input and then printed without any additional processing, the `json` data type outputs the same text that was input, while the `jsonb` data type does not preserve semantically-insignificant details such as whitespace. For example, note the differences here:

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
                      json
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
                      jsonb
-------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

One semantically-insignificant detail worth noting is that with the `jsonb` data type, numbers will be printed according to the behavior of the underlying numeric type. In practice, this means that numbers entered with E notation will be printed without it, for example:

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;
         json          |          jsonb
-----------------------+------------------------
 {"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

However, the `jsonb` data type preserves trailing fractional zeroes, as seen in previous example, even though those are semantically insignificant for purposes such as equality checks.

Designing JSON documents

Representing data as JSON can be considerably more flexible than the traditional relational data model, which is compelling in environments where requirements are fluid. It is quite possible for both approaches to co-exist and complement each other within the same application. However, even for applications where maximal flexibility is desired, it is still recommended that JSON documents have a somewhat fixed structure. The structure is typically unenforced (though enforcing some business rules declaratively is possible), but having a predictable structure makes it easier to write queries that usefully summarize a set of JSON documents (datums) in a table.

JSON data is subject to the same concurrency-control considerations as any other data type when stored in a table. Although storing large documents is practicable, keep in mind that any update acquires a row-level lock on the whole row. Consider limiting JSON documents to a manageable size in order to decrease lock contention among updating transactions. Ideally, JSON documents should each represent an atomic datum that business rules dictate cannot reasonably be further subdivided into smaller datums that could be modified independently.

jsonb Containment and Existence

Testing *containment* is an important capability of `jsonb`. There is no parallel set of facilities for the `json` type. Containment tests whether one `jsonb` document has contained within it another one. These examples return true except as noted:

```
-- Simple scalar/primitive values contain only the identical value:
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

-- The array on the right side is contained within the one on the left:
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- Order of array elements is not significant, so this is also true:
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- Duplicate array elements don't matter either:
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- The object with a single pair on the right side is contained
-- within the object on the left side:
SELECT '{"product": "Greenplum", "version": "6.0.0", "jsonb":true}'::jsonb
 @> '{"version":"6.0.0"}'::jsonb;

-- The array on the right side is not considered contained within the
-- array on the left, even though a similar array is nested within it:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  -- yields false

-- But with a layer of nesting, it is contained:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;
```

```
-- Similarly, containment is not reported here:
SELECT '{"foo": {"bar": "baz", "zig": "zag"}}'::jsonb @> '{"bar":
 "baz"}'::jsonb; -- yields false

-- But with a layer of nesting, it is contained:
SELECT '{"foo": {"bar": "baz", "zig": "zag"}}'::jsonb @> '{"foo": {"bar":
 "baz"}}'::jsonb;
```

The general principle is that the contained object must match the containing object as to structure and data contents, possibly after discarding some non-matching array elements or object key/value pairs from the containing object. For containment, the order of array elements is not significant when doing a containment match, and duplicate array elements are effectively considered only once.

As an exception to the general principle that the structures must match, an array may contain a primitive value:

```
-- This array contains the primitive string value:
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- This exception is not reciprocal -- non-containment is reported here:
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- yields false
```

`jsonb` also has an *existence* operator, which is a variation on the theme of containment: it tests whether a string (given as a text value) appears as an object key or array element at the top level of the `jsonb` value. These examples return true except as noted:

```
-- String exists as array element:
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- String exists as object key:
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- Object values are not considered:
SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- yields false

-- As with containment, existence must match at the top level:
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- yields false

-- A string is considered to exist if it matches a primitive JSON string:
SELECT '"foo"'::jsonb ? 'foo';
```

JSON objects are better suited than arrays for testing containment or existence when there are many keys or elements involved, because unlike arrays they are internally optimized for searching, and do not need to be searched linearly.

The various containment and existence operators, along with all other JSON operators and functions are documented in *JSON Functions and Operators*.

Because JSON containment is nested, an appropriate query can skip explicit selection of sub-objects. As an example, suppose that we have a doc column containing objects at the top level, with most objects containing tags fields that contain arrays of sub-objects. This query finds entries in which sub-objects containing both `"term":"paris"` and `"term":"food"` appear, while ignoring any such keys outside the tags array:

```
SELECT doc->'site_name' FROM websites
  WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';
```

The query with this predicate could accomplish the same thing.

```
SELECT doc->'site_name' FROM websites
  WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';
```

However, the second approach is less flexible and is often less efficient as well.

On the other hand, the JSON existence operator is not nested: it will only look for the specified key or array element at top level of the JSON value.

jsonb Indexing

The Greenplum Database `jsonb` data type, supports GIN, btree, and hash indexes.

- *GIN Indexes on jsonb Data*
- *Btree and Hash Indexes on jsonb Data*

GIN Indexes on jsonb Data

GIN indexes can be used to efficiently search for keys or key/value pairs occurring within a large number of `jsonb` documents (datums). Two GIN operator classes are provided, offering different performance and flexibility trade-offs.

The default GIN operator class for *jsonb* supports queries with the @>, ?, ?& and ?| operators. (For details of the semantics that these operators implement, see the table *Table 51: jsonb Operators*.) An example of creating an index with this operator class is:

```
CREATE INDEX idxgin ON api USING gin (jdoc);
```

The non-default GIN operator class `jsonb_path_ops` supports indexing the @> operator only. An example of creating an index with this operator class is:

```
CREATE INDEX idxginp ON api USING gin (jdoc jsonb_path_ops);
```

Consider the example of a table that stores JSON documents retrieved from a third-party web service, with a documented schema definition. This is a typical document:

```
{
    "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
    "name": "Angela Barton",
    "is_active": true,
    "company": "Magnafone",
    "address": "178 Howard Place, Gulf, Washington, 702",
    "registered": "2009-11-07T08:53:22 +08:00",
    "latitude": 19.793713,
    "longitude": 86.513373,
    "tags": [
        "enim",
        "aliquip",
        "qui"
    ]
}
```

The JSON documents are stored a table named `api`, in a `jsonb` column named `jdoc`. If a GIN index is created on this column, queries like the following can make use of the index:

```
-- Find documents in which the key "company" has value "Magnafone"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company":
 "Magnafone"}';
```

However, the index could not be used for queries like the following. The operator ? is indexable, however, the comparison is not applied directly to the indexed column `jdoc`:

```
-- Find documents in which the key "tags" contains key or array element
 "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

With appropriate use of expression indexes, the above query can use an index. If querying for particular items within the `tags` key is common, defining an index like this might be worthwhile:

```
CREATE INDEX idxgintags ON api USING gin ((jdoc -> 'tags'));
```

Now, the `WHERE` clause `jdoc -> 'tags' ? 'qui'` is recognized as an application of the indexable operator `?` to the indexed expression `jdoc -> 'tags'`. For information about expression indexes, see *Indexes on Expressions*.

Another approach to querying JSON documents is to exploit containment, for example:

```
-- Find documents in which the key "tags" contains array element "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';
```

A simple GIN index on the `jdoc` column can support this query. However, the index will store copies of every key and value in the `jdoc` column, whereas the expression index of the previous example stores only data found under the tags key. While the simple-index approach is far more flexible (since it supports queries about any key), targeted expression indexes are likely to be smaller and faster to search than a simple index.

Although the `jsonb_path_ops` operator class supports only queries with the `@>` operator, it has performance advantages over the default operator class `jsonb_ops`. A `jsonb_path_ops` index is usually much smaller than a `jsonb_ops` index over the same data, and the specificity of searches is better, particularly when queries contain keys that appear frequently in the data. Therefore search operations typically perform better than with the default operator class.

The technical difference between a `jsonb_ops` and a `jsonb_path_ops` GIN index is that the former creates independent index items for each key and value in the data, while the latter creates index items only for each value in the data.

> **Note:** For this discussion, the term *value* includes array elements, though JSON terminology sometimes considers array elements distinct from values within objects.

Basically, each `jsonb_path_ops` index item is a hash of the value and the key(s) leading to it; for example to index `{"foo": {"bar": "baz"}}`, a single index item would be created incorporating all three of `foo`, `bar`, and `baz` into the hash value. Thus a containment query looking for this structure would result in an extremely specific index search; but there is no way at all to find out whether `foo` appears as a key. On the other hand, a `jsonb_ops` index would create three index items representing `foo`, `bar`, and `baz` separately; then to do the containment query, it would look for rows containing all three of these items. While GIN indexes can perform such an `AND` search fairly efficiently, it will still be less specific and slower than the equivalent `jsonb_path_ops` search, especially if there are a very large number of rows containing any single one of the three index items.

A disadvantage of the `jsonb_path_ops` approach is that it produces no index entries for JSON structures not containing any values, such as `{"a": {}}`. If a search for documents containing such a structure is requested, it will require a full-index scan, which is quite slow. `jsonb_path_ops` is ill-suited for applications that often perform such searches.

Btree and Hash Indexes on jsonb Data

`jsonb` also supports `btree` and `hash` indexes. These are usually useful only when it is important to check the equality of complete JSON documents.

For completeness the `btree` ordering for `jsonb` datums is:

```
Object > Array > Boolean > Number > String > Null

Object with n pairs > object with n - 1 pairs

Array with n elements > array with n - 1 elements
```

Objects with equal numbers of pairs are compared in the order:

```
key-1, value-1, key-2 ...
```

Object keys are compared in their storage order. In particular, since shorter keys are stored before longer keys, this can lead to orderings that might not be intuitive, such as:

```
{ "aa": 1, "c": 1} > {"b": 1, "d": 1}
```

Similarly, arrays with equal numbers of elements are compared in the order:

```
element-1, element-2 ...
```

Primitive JSON values are compared using the same comparison rules as for the underlying Greenplum Database data type. Strings are compared using the default database collation.

JSON Functions and Operators

Greenplum Database includes built-in functions and operators that create and manipulate JSON data.

- *JSON Operators*
- *JSON Creation Functions*
- *JSON Aggregate Functions*
- *JSON Processing Functions*

> **Note:** For `json` data type values, all key/value pairs are kept even if a JSON object contains duplicate keys. For duplicate keys, JSON processing functions consider the last value as the operative one. For the `jsonb` data type, duplicate object keys are not kept. If the input includes duplicate keys, only the last value is kept. See *About JSON Data*.

JSON Operators

This table describes the operators that are available for use with the `json` and `jsonb` data types.

Table 50: *json* and `jsonb` Operators

Operator	Right Operand Type	Description	Example	Example Result
`->`	`int`	Get the JSON array element (indexed from zero).	`'[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->2`	`{"c":"baz"}`
`->`	`text`	Get the JSON object field by key.	`'{"a": {"b":"foo"}}'::json->'a'`	`{"b":"foo"}`
`->>`	`int`	Get the JSON array element as `text`.	`'[1,2,3]'::json->>2`	`3`
`->>`	`text`	Get the JSON object field as `text`.	`'{"a":1,"b":2}'::json->>'b'`	`2`
`#>`	`text[]`	Get the JSON object at specified path.	`'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}'`	`{"c": "foo"}`
`#>>`	`text[]`	Get the JSON object at specified path as `text`.	`'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'`	`3`

> **Note:** There are parallel variants of these operators for both the `json` and `jsonb` data types. The field, element, and path extraction operators return the same data type as their left-hand input (either `json` or `jsonb`), except for those specified as returning `text`, which coerce the value to

`text`. The field, element, and path extraction operators return `NULL`, rather than failing, if the JSON input does not have the right structure to match the request; for example if no such element exists.

Operators that require the `jsonb` data type as the left operand are described in the following table. Many of these operators can be indexed by `jsonb` operator classes. For a full description of `jsonb` containment and existence semantics, see *jsonb Containment and Existence*. For information about how these operators can be used to effectively index `jsonb` data, see *jsonb Indexing*.

Table 51: `jsonb` Operators

Operator	Right Operand Type	Description	Example
@>	jsonb	Does the left JSON value contain within it the right value?	'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb
<@	jsonb	Is the left JSON value contained within the right value?	'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb
?	text	Does the key/element string exist within the JSON value?	'{"a":1, "b":2}'::jsonb ? 'b'
?|	text[]	Do any of these key/element strings exist?	'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']
?&	text[]	Do all of these key/element strings exist?	'["a", "b"]'::jsonb ?& array['a', 'b']

The standard comparison operators in the following table are available only for the `jsonb` data type, not for the `json` data type. They follow the ordering rules for B-tree operations described in *jsonb Indexing*.

Table 52: `jsonb` Comparison Operators

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
<> or !=	not equal

Note: The != operator is converted to <> in the parser stage. It is not possible to implement != and <> operators that do different things.

JSON Creation Functions

This table describes the functions that create `json` data type values. (Currently, there are no equivalent functions for `jsonb`, but you can cast the result of one of these functions to `jsonb`.)

Table 53: JSON Creation Functions

Function	Description	Example	Example Result
`to_json(anyelement)`	Returns the value as a JSON object. Arrays and composites are processed recursively and are converted to arrays and objects. If the input contains a cast from the type to `json`, the cast function is used to perform the conversion; otherwise, a JSON scalar value is produced. For any scalar type other than a number, a Boolean, or a null value, the text representation will be used, properly quoted and escaped so that it is a valid JSON string.	`to_json('Fred said "Hi."'::text)`	`"Fred said \"Hi.\""`
`array_to_json(anyarray [, pretty_bool])`	Returns the array as a JSON array. A multidimensional array becomes a JSON array of arrays. Line feeds will be added between dimension-1 elements if `pretty_bool` is true.	`array_to_json('{{1,5},{99,100}}'::int[])`	`[[1,5],[99,100]]`
`row_to_json(record [, pretty_bool])`	Returns the row as a JSON object. Line feeds will be added between level-1 elements if `pretty_bool` is true.	`row_to_json(row(1,'foo'))`	`{"f1":1,"f2":"foo"}`
`json_build_array(VARIADIC "any")`	Builds a possibly-heterogeneously-typed JSON array out of a `VARIADIC` argument list.	`json_build_array(1,2,'3',4,5)`	`[1, 2, "3", 4, 5]`
`json_build_object(VARIADIC "any")`	Builds a JSON object out of a `VARIADIC` argument list. The argument list is taken in order and converted to a set of key/value pairs.	`json_build_object('foo',1,'bar',2)`	`{"foo": 1, "bar": 2}`

Function	Description	Example	Example Result
`json_object(text[])`	Builds a JSON object out of a text array. The array must be either a one or a two dimensional array. The one dimensional array must have an even number of elements. The elements are taken as key/value pairs. For a two dimensional array, each inner array must have exactly two elements, which are taken as a key/value pair.	`json_object('{a, 1, b, "def", c, 3.5}')` `json_object('{{a, 1},{b, "def"},{c, 3.5}}')`	`{"a": "1", "b": "def", "c": "3.5"}`
`json_object(keys text[], values text[])`	Builds a JSON object out of a text array. This form of `json_object` takes keys and values pairwise from two separate arrays. In all other respects it is identical to the one-argument form.	`json_object('{a, b}', '{1,2}')`	`{"a": "1", "b": "2"}`

Note: `array_to_json` and `row_to_json` have the same behavior as `to_json` except for offering a pretty-printing option. The behavior described for `to_json` likewise applies to each individual value converted by the other JSON creation functions.

Note: The *hstore module* contains functions that cast from `hstore` to `json`, so that `hstore` values converted via the JSON creation functions will be represented as JSON objects, not as primitive string values.

JSON Aggregate Functions

This table shows the functions aggregate records to an array of JSON objects and pairs of values to a JSON object

Table 54: JSON Aggregate Functions

Function	Argument Types	Return Type	Description
`json_agg(record)`	`record`	`json`	Aggregates records as a JSON array of objects.
`json_object_agg(name, value)`	`("any", "any")`	`json`	Aggregates name/value pairs as a JSON object.

JSON Processing Functions

This table shows the functions that are available for processing `json` and `jsonb` values.

Many of these processing functions and operators convert Unicode escapes in JSON strings to the appropriate single character. This is a not an issue if the input data type is `jsonb`, because the conversion

was already done. However, for `json` data type input, this might result in an error being thrown. See *About JSON Data*.

Table 55: JSON Processing Functions

Function	Return Type	Description	Example	Example Result
`json_array_length(json)` `jsonb_array_length(jsonb)`	`int`	Returns the number of elements in the outermost JSON array.	`json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]')`	`5`

`json_each(json)`
`jsonb_each(jsonb)`
	`setof key text, value json`
	`setof key text, value jsonb`
		Expands the outermost JSON object into a set of key/value pairs.	`select * from json_each('{"a":"foo", "b":"bar"}')`	`key | value`
				`-----+-------`
				`a | "foo"`
				`b | "bar"`

`json_each_text(json)`
`jsonb_each_text(jsonb)`
	`setof key text, value text`	Expands the outermost JSON object into a set of key/value pairs. The returned values will be of type `text`.	`select * from json_each_text('{"a":"foo", "b":"bar"}')`	`key | value`
				`-----+-------`
				`a | foo`
				`b | bar`

`json_extract_path(from_json json, VARIADIC path_elems text[])` `jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])`	`json` `jsonb`	Returns the JSON value pointed to by `path_elems` (equivalent to #> operator).	`json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4')`	`{"f5":99,"f6":"foo"}`
`json_extract_path_text(from_json json, VARIADIC path_elems text[])` `jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])`	`text`	Returns the JSON value pointed to by `path_elems` as text. Equivalent to #>> operator.	`json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4', 'f6')`	`foo`

Function	Return Type	Description	Example	Example Result

`json_object_keys(json)`
`jsonb_object_keys(jsonb)`
	`setof text`	Returns set of keys in the outermost JSON object.	`json_object_keys('{"f1":"abc",{"f3":"a", "f4":"b"}}')`	`json_object_keys`
				`----------`
				`f1`
				`f2`

`json_populate_record(base anyelement, from_json json)`
`jsonb_populate_record(base anyelement, from_jsonb jsonb)`
	`anyelement`	Expands the object in `from_json` to a row whose columns match the record type defined by base. See *Note 1*.	`select * from json_populate_record(null::myrow '{"a":1,"b":2}')`	`a | b`
				`---+---`
				`1 | 2`

`json_populate_recordset(base anyelement, from_json json)`
`jsonb_populate_recordset(base anyelement, from_jsonb jsonb)`
	`setof anyelement`	Expands the outermost array of objects in `from_json` to a set of rows whose columns match the record type defined by base. See *Note 1*.	`select * from json_populate_recordset(null::my '[{"a":1,"b":2}, {"a":3,"b":4}]')`	`a | b`
				`---+---`
				`1 | 2`
				`3 | 4`

`json_array_elements(json)`
`jsonb_array_elements(jsonb)`
	`setof json`
	`setof jsonb`
		Expands a JSON array to a set of JSON values.	`select * from json_array_elements('[1,true, [2,false]]')`	`value`
				`-----------`
				`1`
				`true`
				`[2,false]`

`json_array_elements_text(json)`
`jsonb_array_elements_text(jsonb)`
	`setof text`	Expands a JSON array to a set of `text` values.	`select * from json_array_elements_text('["foo", "bar"]')`	`value`
				`-----------`
				`foo`
				`bar`

`json_typeof(json)` `jsonb_typeof(jsonb)`	`text`	Returns the type of the outermost JSON value as a text string. Possible types are `object`, `array`, `string`, `number`, `boolean`, and `null`. See *Note 2*.	`json_typeof('-123.4')`	`number`

Function	Return Type	Description	Example	Example Result

`json_to_record(json)`
`jsonb_to_record(jsonb)`
	`record`	Builds an arbitrary record from a JSON object. See *Note 1*.
		As with all functions returning record, the caller must explicitly define the structure of the record with an AS clause.
			`select * from json_to_record('{"a":1,"b":[1,2,3],"c":"bar"}') as x(a int, b text, d text)`	`a | b | d`
				`---+---------+---`
				`1 | [1,2,3] |`

`json_to_recordset(json)`
`jsonb_to_recordset(jsonb)`
	`setof record`	Builds an arbitrary set of records from a JSON array of objects See *Note 1*.
		As with all functions returning record, the caller must explicitly define the structure of the record with an AS clause.
			`select * from json_to_recordset('[{"a":1,"b":"foo"},{"a":"2","c":"bar"}]') as x(a int, b text);`	`a | b`
				`---+-----`
				`1 | foo`
				`2 |`

Note:

1. The examples for the functions `json_populate_record()`, `json_populate_recordset()`, `json_to_record()` and `json_to_recordset()` use constants. However, the typical use would be to reference a table in the `FROM` clause and use one of its `json` or `jsonb` columns as an argument to the function. The extracted key values can then be referenced in other parts of the query. For example the value can be referenced in `WHERE` clauses and target lists. Extracting multiple values in this way can improve performance over extracting them separately with per-key operators.

   JSON keys are matched to identical column names in the target row type. JSON type coercion for these functions might not result in desired values for some types. JSON fields that do not appear in the target row type will be omitted from the output, and target columns that do not match any JSON field will be `NULL`.
2. The `json_typeof` function null return value of `null` should not be confused with a SQL `NULL`. While calling `json_typeof('null'::json)` will return `null`, calling `json_typeof(NULL::json)` will return a SQL `NULL`.

Working with XML Data

Greenplum Database supports the `xml` data type that stores XML data.

The `xml` data type checks the input values for well-formedness, providing an advantage over simply storing XML data in a text field. Additionally, support functions allow you to perform type-safe operations on this data; refer to *XML Function Reference*, below.

The `xml` type can store well-formed "documents", as defined by the XML standard, as well as "content" fragments, which are defined by reference to the more permissive *document node* of the XQuery and

XPath model. Roughly, this means that content fragments can have more than one top-level element or character node. The expression *xmlvalue* `IS DOCUMENT` can be used to evaluate whether a particular `xml` value is a full document or only a content fragment.

This section contains the following topics:

Creating XML Values

To produce a value of type `xml` from character data, use the function `xmlparse`:

```
xmlparse ( { DOCUMENT | CONTENT } value)
```

For example:

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</
title><chapter>...</chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

The above method converts character strings into XML values according to the SQL standard, but you can also use Greenplum Database syntax like the following:

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

The `xml` type does not validate input values against a document type declaration (DTD), even when the input value specifies a DTD. There is also currently no built-in support for validating against other XML schema languages such as XML schema.

The inverse operation, producing a character string value from `xml`, uses the function `xmlserialize`:

```
xmlserialize ( { DOCUMENT | CONTENT } value AS type )
```

type can be `character`, `character varying`, or `text` (or an alias for one of those). Again, according to the SQL standard, this is the only way to convert between type `xml` and character types, but Greenplum Database also allows you to simply cast the value.

When a character string value is cast to or from type `xml` without going through `XMLPARSE` or `XMLSERIALIZE`, respectively, the choice of `DOCUMENT` versus `CONTENT` is determined by the `XML OPTION` session configuration parameter, which can be set using the standard command:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

or simply like Greenplum Database:

```
SET XML OPTION TO { DOCUMENT | CONTENT };
```

The default is CONTENT, so all forms of XML data are allowed.

Encoding Handling

Be careful when dealing with multiple character encodings on the client, server, and in the XML data passed through them. When using the text mode to pass queries to the server and query results to the

client (which is the normal mode), Greenplum Database converts all character data passed between the client and the server, and vice versa, to the character encoding of the respective endpoint; see *Character Set Support*. This includes string representations of XML values, such as in the above examples. Ordinarily, this means that encoding declarations contained in XML data can become invalid, as the character data is converted to other encodings while travelling between client and server, because the embedded encoding declaration is not changed. To cope with this behavior, encoding declarations contained in character strings presented for input to the `xml` type are ignored, and content is assumed to be in the current server encoding. Consequently, for correct processing, character strings of XML data must be sent from the client in the current client encoding. It is the responsibility of the client to either convert documents to the current client encoding before sending them to the server, or to adjust the client encoding appropriately. On output, values of type `xml` will not have an encoding declaration, and clients should assume all data is in the current client encoding.

When using binary mode to pass query parameters to the server and query results back to the client, no character set conversion is performed, so the situation is different. In this case, an encoding declaration in the XML data will be observed, and if it is absent, the data will be assumed to be in UTF-8 (as required by the XML standard; note that Greenplum Database does not support UTF-16). On output, data will have an encoding declaration specifying the client encoding, unless the client encoding is UTF-8, in which case it will be omitted.

Note:

Processing XML data with Greenplum Database will be less error-prone and more efficient if the XML data encoding, client encoding, and server encoding are the same. Because XML data is internally processed in UTF-8, computations will be most efficient if the server encoding is also UTF-8.

Accessing XML Values

The `xml` data type is unusual in that it does not provide any comparison operators. This is because there is no well-defined and universally useful comparison algorithm for XML data. One consequence of this is that you cannot retrieve rows by comparing an `xml` column against a search value. XML values should therefore typically be accompanied by a separate key field such as an ID. An alternative solution for comparing XML values is to convert them to character strings first, but note that character string comparison has little to do with a useful XML comparison method.

Because there are no comparison operators for the `xml` data type, it is not possible to create an index directly on a column of this type. If speedy searches in XML data are desired, possible workarounds include casting the expression to a character string type and indexing that, or indexing an XPath expression. Of course, the actual query would have to be adjusted to search by the indexed expression.

Processing XML

To process values of data type `xml`, Greenplum Database offers the functions `xpath` and `xpath_exists`, which evaluate XPath 1.0 expressions.

```
xpath(xpath, xml [, nsarray])
```

The function `xpath` evaluates the XPath expression *`xpath`* (a text value) against the XML value *`xml`*. It returns an array of XML values corresponding to the node set produced by the XPath expression.

The second argument must be a well formed XML document. In particular, it must have a single root node element.

The optional third argument of the function is an array of namespace mappings. This array should be a two-dimensional text array with the length of the second axis being equal to 2 (i.e., it should be an array of arrays, each of which consists of exactly 2 elements). The first element of each array entry is the namespace name (alias), the second the namespace URI. It is not required that aliases provided in

this array be the same as those being used in the XML document itself (in other words, both in the XML document and in the `xpath` function context, aliases are local).

Example:

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</
my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

To deal with default (anonymous) namespaces, do something like this:

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</
b></a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

```
xpath_exists(xpath, xml [, nsarray])
```

The function `xpath_exists` is a specialized form of the `xpath` function. Instead of returning the individual XML values that satisfy the XPath, this function returns a Boolean indicating whether the query was satisfied or not. This function is equivalent to the standard `XMLEXISTS` predicate, except that it also offers support for a namespace mapping argument.

Example:

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://
example.com">test</my:a>',
                    ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
--------------
 t
(1 row)
```

Mapping Tables to XML

The following functions map the contents of relational tables to XML values. They can be thought of as XML export functionality:

```
table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns
 text)
query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int, nulls boolean,
              tableforest boolean, targetns text)
```

The return type of each function is `xml`.

`table_to_xml` maps the content of the named table, passed as parameter `tbl`. The `regclass` type accepts strings identifying tables using the usual notation, including optional schema qualifications and double quotes. `query_to_xml` executes the query whose text is passed as parameter query and maps the result set. `cursor_to_xml` fetches the indicated number of rows from the cursor specified by the

parameter cursor. This variant is recommended if large tables have to be mapped, because the result value is built up in memory by each function.

If `tableforest` is false, then the resulting XML document looks like this:

```
<tablename>
  <row>
    <columnname1>data</columnname1>
    <columnname2>data</columnname2>
  </row>

  <row>
    ...
  </row>

  ...
</tablename>
```

If `tableforest` is true, the result is an XML content fragment that looks like this:

```
<tablename>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
</tablename>

<tablename>
  ...
</tablename>

...
```

If no table name is available, that is, when mapping a query or a cursor, the string `table` is used in the first format, `row` in the second format.

The choice between these formats is up to the user. The first format is a proper XML document, which will be important in many applications. The second format tends to be more useful in the `cursor_to_xml` function if the result values are to be later reassembled into one document. The functions for producing XML content discussed above, in particular `xmlelement`, can be used to alter the results as desired.

The data values are mapped in the same way as described for the function `xmlelement`, above.

The parameter nulls determines whether null values should be included in the output. If true, null values in columns are represented as:

```
<columnname xsi:nil="true"/>
```

where `xsi` is the XML namespace prefix for XML schema Instance. An appropriate namespace declaration will be added to the result value. If false, columns containing null values are simply omitted from the output.

The parameter `targetns` specifies the desired XML namespace of the result. If no particular namespace is wanted, an empty string should be passed.

The following functions return XML schema documents describing the mappings performed by the corresponding functions above:

```
able_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns
 text)
query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns
 text)
cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean,
 targetns text)
```

It is essential that the same parameters are passed in order to obtain matching XML data mappings and XML schema documents.

The following functions produce XML data mappings and the corresponding XML schema in one document (or `forest`), linked together. They can be useful where self-contained and self-describing results are desired:

```
table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean,
 targetns text)
query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean,
 targetns text)
```

In addition, the following functions are available to produce analogous mappings of entire schemas or the entire current database:

```
schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns
 text)
schema_to_xmlschema(schema name, nulls boolean, tableforest boolean,
 targetns text)
schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean,
 targetns text)

database_to_xml(nulls boolean, tableforest boolean, targetns text)
database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns
 text)
```

Note that these potentially produce large amounts of data, which needs to be built up in memory. When requesting content mappings of large schemas or databases, consider mapping the tables separately instead, possibly even through a cursor.

The result of a schema content mapping looks like this:

```
<schemaname>

table1-mapping

table2-mapping

...

</schemaname>
```

where the format of a table mapping depends on the `tableforest` parameter, as explained above.

The result of a database content mapping looks like this:

```
<dbname>

<schema1name>
  ...
</schema1name>

<schema2name>
  ...
</schema2name>

...

</dbname>
```

where the schema mapping is as above.

The example below demonstrates using the output produced by these functions, The example shows an XSLT stylesheet that converts the output of `table_to_xml_and_xmlschema` to an HTML document containing a tabular rendition of the table data. In a similar manner, the results from these functions can be converted into other XML-based formats.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/xsd:element[@name=name(current())]/@type"/
>
    <xsl:variable name="rowtypename"
                  select="$schema/xsd:complexType[@name=$tabletypename]/
xsd:sequence/xsd:element[@name='row']/@type"/>

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=
$rowtypename]/xsd:sequence/xsd:element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

XML Function Reference

The functions described in this section operate on values of type `xml`. The section *XML Predicates*also contains information about the `xml` functions and function-like expressions.

Function:

`xmlcomment`

Synopsis:

```
xmlcomment(text)
```

The function `xmlcomment` creates an XML value containing an XML comment with the specified text as content. The text cannot contain "--" or end with a "-" so that the resulting construct is a valid XML comment. If the argument is null, the result is null.

Example:

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

Function:

`xmlconcat`

Synopsis:

```
xmlconcat(xml[, …])
```

The function `xmlconcat` concatenates a list of individual XML values to create a single value containing an XML content fragment. Null values are omitted; the result is only null if there are no nonnull arguments.

Example:

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

XML declarations, if present, are combined as follows:

- If all argument values have the same XML version declaration, that version is used in the result, else no version is used.
- If all argument values have the standalone declaration value "yes", then that value is used in the result.
- If all argument values have a standalone declaration value and at least one is "no", then that is used in the result. Otherwise, the result will have no standalone declaration.
- If the result is determined to require a standalone declaration but no version declaration, a version declaration with version 1.0 will be used because XML requires an XML declaration to contain a version declaration.

Encoding declarations are ignored and removed in all cases.

Example:

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1"
 standalone="no"?><bar/>');

             xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

Function:

`xmlelement`

Synopsis:

```
xmlelement(name name [, xmlattributes(value [AS attname] [, ... ])] [,
 content, ...])
```

The `xmlelement` expression produces an XML element with the given name, attributes, and content.

Examples:

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

    xmlelement
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont',
 'ent');

             xmlelement
-------------------------------------
 <foo bar="2017-01-26">content</foo>
```

Element and attribute names that are not valid XML names are escaped by replacing the offending characters by the sequence _x*HHHH*_, where *HHHH* is the character's Unicode codepoint in hexadecimal notation. For example:

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

            xmlelement
----------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

An explicit attribute name need not be specified if the attribute value is a column reference, in which case the column's name will be used as the attribute name by default. In other cases, the attribute must be given an explicit name. So this example is valid:

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

But these are not:

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

Element content, if specified, will be formatted according to its data type. If the content is itself of type `xml`, complex XML documents can be constructed. For example:

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                            xmlelement(name abc),
                            xmlcomment('test'),
                            xmlelement(name xyz));

                  xmlelement
----------------------------------------------
```

```
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

Content of other types will be formatted into valid XML character data. This means in particular that the characters <, >, and & will be converted to entities. Binary data (data type `bytea`) will be represented in base64 or hex encoding, depending on the setting of the configuration parameter *xmlbinary*. The particular behavior for individual data types is expected to evolve in order to align the SQL and Greenplum Database data types with the XML schema specification, at which point a more precise description will appear.

Function:

`xmlforest`

Synopsis:

```
xmlforest(content [AS name] [, ...])
```

The `xmlforest` expression produces an XML forest (sequence) of elements using the given names and content.

Examples:

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

          xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>


SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

                                          xmlforest
-----------------------------------------------------------------------------------------
 <table_name>pg_authid</table_name><column_name>rolname</column_name>
 <table_name>pg_authid</table_name><column_name>rolsuper</column_name>
```

As seen in the second example, the element name can be omitted if the content value is a column reference, in which case the column name is used by default. Otherwise, a name must be specified.

Element names that are not valid XML names are escaped as shown for `xmlelement` above. Similarly, content data is escaped to make valid XML content, unless it is already of type `xml`.

Note that XML forests are not valid XML documents if they consist of more than one element, so it might be useful to wrap `xmlforest` expressions in `xmlelement`.

Function:

`xmlpi`

Synopsis:

```
xmlpi(name target [, content])
```

The `xmlpi` expression creates an XML processing instruction. The content, if present, must not contain the character sequence ?>.

Example:

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
```

```
 <?php echo "hello world";?>
```

Function:

`xmlroot`

Synopsis:

```
xmlroot(xml, version text | no value [, standalone yes|no|no value])
```

The `xmlroot` expression alters the properties of the root node of an XML value. If a version is specified, it replaces the value in the root node's version declaration; if a standalone setting is specified, it replaces the value in the root node's standalone declaration.

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</
content>'),
               version '1.0', standalone yes);

                xmlroot
----------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

Function:

`xmlagg`

```
xmlagg (xml)
```

The function `xmlagg` is, unlike the other functions described here, an aggregate function. It concatenates the input values to the aggregate function call, much like `xmlconcat` does, except that concatenation occurs across rows rather than across expressions in a single row. See *Using Functions and Operators* for additional information about aggregate functions.

Example:

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
        xmlagg
----------------------
 <foo>abc</foo><bar/>
```

To determine the order of the concatenation, an `ORDER BY` clause may be added to the aggregate call. For example:

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

The following non-standard approach used to be recommended in previous versions, and may still be useful in specific cases:

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

XML Predicates

The expressions described in this section check properties of `xml` values.

Expression:

`IS DOCUMENT`

Synopsis:

```
xml IS DOCUMENT
```

The expression `IS DOCUMENT` returns true if the argument XML value is a proper XML document, false if it is not (that is, it is a content fragment), or null if the argument is null.

Expression:

`XMLEXISTS`

Synopsis:

```
XMLEXISTS(text PASSING [BY REF] xml [BY REF])
```

The function `xmlexists` returns true if the XPath expression in the first argument returns any nodes, and false otherwise. (If either argument is null, the result is null.)

Example:

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY REF
 '<towns><town>Toronto</town><town>Ottawa</town></towns>');

 xmlexists
------------
 t
(1 row)
```

The `BY REF` clauses have no effect in Greenplum Database, but are allowed for SQL conformance and compatibility with other implementations. Per SQL standard, the first `BY REF` is required, the second is optional. Also note that the SQL standard specifies the `xmlexists` construct to take an XQuery expression as first argument, but Greenplum Database currently only supports XPath, which is a subset of XQuery.

Expression:

`xml_is_well_formed`

Synopsis:

```
xml_is_well_formed(text)
xml_is_well_formed_document(text)
xml_is_well_formed_content(text)
```

These functions check whether a text string is well-formed XML, returning a Boolean result. `xml_is_well_formed_document` checks for a well-formed document, while `xml_is_well_formed_content` checks for well-formed content. `xml_is_well_formed` does the former if the `xmloption` configuration parameter is set to `DOCUMENT`, or the latter if it is set to `CONTENT`. This means that `xml_is_well_formed` is useful for seeing whether a simple cast to type `xml` will succeed, whereas the other two functions are useful for seeing whether the corresponding variants of `XMLPARSE` will succeed.

Examples:

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/
stuff">bar</pg:foo>');
 xml_is_well_formed_document
-----------------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/
stuff">bar</my:foo>');
 xml_is_well_formed_document
-----------------------------
 f
(1 row)
```

The last example shows that the checks include whether namespaces are correctly matched.

Using Full Text Search

Greenplum Database provides data types, functions, operators, index types, and configurations for querying natural language documents.

About Full Text Search

This topic provides an overview of Greenplum Database full text search, basic text search expressions, configuring, and customizing text search. Greenplum Database full text search is compared with Pivotal GPText.

This section contains the following subtopics:

- *What is a Document?*
- *Basic Text Matching*
- *Configurations*
- *Comparing Greenplum Database Text Search with Pivotal GPText*

Full Text Searching (or just "text search") provides the capability to identify natural-language *documents* that satisfy a *query*, and optionally to rank them by relevance to the query. The most common type of search is to find all documents containing given *query terms* and return them in order of their *similarity* to the query.

Greenplum Database provides a data type `tsvector` to store preprocessed documents, and a data type `tsquery` to store processed queries (*Text Search Data Types*). There are many functions and operators available for these data types (*Text Search Functions and Operators*), the most important of which is the match operator `@@`, which we introduce in *Basic Text Matching*. Full text searches can be accelerated using indexes (*GiST and GIN Indexes for Text Search*).

Notions of query and similarity are very flexible and depend on the specific application. The simplest search considers query as a set of words and similarity as the frequency of query words in the document.

Greenplum Database supports the standard text matching operators `~`, `~*`, `LIKE`, and `ILIKE` for textual data types, but these operators lack many essential properties required for searching documents:

- There is no linguistic support, even for English. Regular expressions are not sufficient because they cannot easily handle derived words, e.g., `satisfies` and `satisfy`. You might miss documents that contain `satisfies`, although you probably would like to find them when searching for `satisfy`. It is possible to use OR to search for multiple derived forms, but this is tedious and error-prone (some words can have several thousand derivatives).
- They provide no ordering (ranking) of search results, which makes them ineffective when thousands of matching documents are found.
- They tend to be slow because there is no index support, so they must process all documents for every search.

Full text indexing allows documents to be preprocessed and an index saved for later rapid searching. Preprocessing includes:

- **Parsing documents into tokens.** It is useful to identify various classes of tokens, e.g., numbers, words, complex words, email addresses, so that they can be processed differently. In principle token classes depend on the specific application, but for most purposes it is adequate to use a predefined set of classes. Greenplum Database uses a *parser* to perform this step. A standard parser is provided, and custom parsers can be created for specific needs.
- **Converting tokens into lexemes.** A lexeme is a string, just like a token, but it has been *normalized* so that different forms of the same word are made alike. For example, normalization almost always includes folding upper-case letters to lower-case, and often involves removal of suffixes (such as s or es in English). This allows searches to find variant forms of the same word, without tediously entering all the possible variants. Also, this step typically eliminates *stop words*, which are words that are so common that they are useless for searching. (In short, then, tokens are raw fragments of the document text, while lexemes are words that are believed useful for indexing and searching.) Greenplum Database uses *dictionaries* to perform this step. Various standard dictionaries are provided, and custom ones can be created for specific needs.
- **Storing preprocessed documents optimized for searching.** For example, each document can be represented as a sorted array of normalized lexemes. Along with the lexemes it is often desirable to store positional information to use for *proximity ranking*, so that a document that contains a more "dense" region of query words is assigned a higher rank than one with scattered query words.

Dictionaries allow fine-grained control over how tokens are normalized. With appropriate dictionaries, you can:

- Define stop words that should not be indexed.
- Map synonyms to a single word using Ispell.
- Map phrases to a single word using a thesaurus.
- Map different variations of a word to a canonical form using an Ispell dictionary.
- Map different variations of a word to a canonical form using Snowball stemmer rules.

What is a Document?

A *document* is the unit of searching in a full text search system; for example, a magazine article or email message. The text search engine must be able to parse documents and store associations of lexemes (key words) with their parent document. Later, these associations are used to search for documents that contain query words.

For searches within Greenplum Database, a document is normally a textual field within a row of a database table, or possibly a combination (concatenation) of such fields, perhaps stored in several tables or obtained dynamically. In other words, a document can be constructed from different parts for indexing and it might not be stored anywhere as a whole. For example:

```
SELECT title || ' ' ||  author || ' ' ||  abstract || ' ' || body AS
 document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS
 document
FROM messages m, docs d
WHERE mid = did AND mid = 12;
```

Note:

Actually, in these example queries, `coalesce` should be used to prevent a single `NULL` attribute from causing a `NULL` result for the whole document.

Another possibility is to store the documents as simple text files in the file system. In this case, the database can be used to store the full text index and to execute searches, and some unique identifier can be used to retrieve the document from the file system. However, retrieving files from outside the database requires superuser permissions or special function support, so this is usually less convenient than keeping all the data inside Greenplum Database. Also, keeping everything inside the database allows easy access to document metadata to assist in indexing and display.

For text search purposes, each document must be reduced to the preprocessed `tsvector` format. Searching and ranking are performed entirely on the tsvector representation of a document — the original text need only be retrieved when the document has been selected for display to a user. We therefore often speak of the `tsvector` as being the document, but of course it is only a compact representation of the full document.

Basic Text Matching

Full text searching in Greenplum Database is based on the match operator @@, which returns `true` if a `tsvector` (document) matches a `tsquery` (query). It does not matter which data type is written first:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat &
 rat'::tsquery;
 ?column?
----------
 t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat
 rat'::tsvector;
 ?column?
----------
 f
```

As the above example suggests, a `tsquery` is not just raw text, any more than a `tsvector` is. A `tsquery` contains search terms, which must be already-normalized lexemes, and may combine multiple terms using AND, OR, and NOT operators. (For details see.) There are functions `to_tsquery` and `plainto_tsquery` that are helpful in converting user-written text into a proper `tsquery`, for example by normalizing words appearing in the text. Similarly, `to_tsvector` is used to parse and normalize a document string. So in practice a text search match would look more like this:

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
 ?column?
----------
```

```
 t
```

Observe that this match would not succeed if written as

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
 ?column?
----------
 f
```

since here no normalization of the word `rats` will occur. The elements of a `tsvector` are lexemes, which are assumed already normalized, so `rats` does not match `rat`.

The @@ operator also supports `text` input, allowing explicit conversion of a text string to `tsvector` or `tsquery` to be skipped in simple cases. The variants available are:

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

The first two of these we saw already. The form `text @@ tsquery` is equivalent to `to_tsvector(x) @@ y`. The form `text @@ text` is equivalent to `to_tsvector(x) @@ plainto_tsquery(y)`.

Configurations

The above are all simple text search examples. As mentioned before, full text search functionality includes the ability to do many more things: skip indexing certain words (stop words), process synonyms, and use sophisticated parsing, e.g., parse based on more than just white space. This functionality is controlled by *text search configurations*. Greenplum Database comes with predefined configurations for many languages, and you can easily create your own configurations. (psql's `\dF` command shows all available configurations.)

During installation an appropriate configuration is selected and *default_text_search_config* is set accordingly in `postgresql.conf`. If you are using the same text search configuration for the entire cluster you can use the value in `postgresql.conf`. To use different configurations throughout the cluster but the same configuration within any one database, use `ALTER DATABASE ... SET`. Otherwise, you can set `default_text_search_config` in each session.

Each text search function that depends on a configuration has an optional `regconfig` argument, so that the configuration to use can be specified explicitly. `default_text_search_config` is used only when this argument is omitted.

To make it easier to build custom text search configurations, a configuration is built up from simpler database objects. Greenplum Database's text search facility provides four types of configuration-related database objects:

- *Text search parsers* break documents into tokens and classify each token (for example, as words or numbers).
- *Text search dictionaries* convert tokens to normalized form and reject stop words.
- *Text search templates* provide the functions underlying dictionaries. (A dictionary simply specifies a template and a set of parameters for the template.)
- *Text search configurations* select a parser and a set of dictionaries to use to normalize the tokens produced by the parser.

Text search parsers and templates are built from low-level C functions; therefore it requires C programming ability to develop new ones, and superuser privileges to install one into a database. (There are examples of add-on parsers and templates in the `contrib/` area of the Greenplum Database distribution.) Since dictionaries and configurations just parameterize and connect together some underlying parsers and templates, no special privilege is needed to create a new dictionary or configuration. Examples of creating custom dictionaries and configurations appear later in this chapter.

Comparing Greenplum Database Text Search with Pivotal GPText

Greenplum Database text search is PostgreSQL text search ported to the Greenplum Database MPP platform. Pivotal also offers Pivotal GPText, which integrates Greenplum Database with the Apache Solr text search platform. GPText installs an Apache Solr cluster alongside your Greenplum Database cluster and provides Greenplum Database functions you can use to create Solr indexes, query them, and receive results in the database session.

Both of these systems provide powerful, enterprise-quality document indexing and searching services. Greenplum Database text search is immediately available to you, with no need to install and maintain additional software. If it meets your applications' requirements, you should use it.

GPText, with Solr, has many capabilities that are not available with Greenplum Database text search. In particular, GPText is better for advanced text analysis applications. Following are some of the advantages and capabilities available to you when you use GPText for text search applications.

- The Apache Solr cluster can be scaled separately from the database. Solr nodes can be deployed on the Greenplum Database hosts or on separate hosts on the network.
- Indexing and search workloads can be moved out of Greenplum Database to Solr to maintain database query performance.
- GPText creates Solr indexes that are split into *shards*, one per Greenplum Database segment, so the advantages of the Greenplum Database MPP architecture are extended to text search workloads.
- Indexing and searching documents with Solr is very fast and can be scaled by adding more Solr nodes to the cluster.
- Document content can be stored in Greenplum Database tables, in the Solr index, or both.
- Through GPText, Solr can index documents stored as text in Greenplum Database tables, as well as documents in external stores accessible using HTTP, FTP, S3, or HDFS URLs.
- Solr automatically recognizes most rich document formats and indexes document content and metadata separately.
- Solr indexes are highly customizable. You can customize the text analysis chain down to the field level.
- In addition to the large number of languages, tokenizers, and filters available from the Apache project, GPText provides a social media tokenizer, an international text tokenizer, and a universal query parser that understands several common text search syntaxes.
- The GPText API supports advanced text analysis tools, such as facetting, named entity recognition (NER), and parts of speech (POS) recognition.

See the *GPText Documentation web site* for more information about GPText.

Searching Text in Database Tables

This topic shows how to use text search operators to search database tables and how to create indexes to speed up text searches.

The examples in the previous section illustrated full text matching using simple constant strings. This section shows how to search table data, optionally using indexes.

This section contains the following subtopics:

- *Searching a Table*
- *Creating Indexes*

Searching a Table

It is possible to do a full text search without an index. A simple query to print the `title` of each row that contains the word `friend` in its `body` field is:

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

This will also find related words such as `friends` and `friendly`, since all these are reduced to the same normalized lexeme.

The query above specifies that the `english` configuration is to be used to parse and normalize the strings. Alternatively we could omit the configuration parameters:

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

This query will use the configuration set by *default_text_search_config*.

A more complex example is to select the ten most recent documents that contain `create` and `table` in the `title` or `body`:

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

For clarity we omitted the `coalesce` function calls which would be needed to find rows that contain `NULL` in one of the two fields.

Although these queries will work without an index, most applications will find this approach too slow, except perhaps for occasional ad-hoc searches. Practical use of text searching usually requires creating an index.

Creating Indexes

We can create a GIN index (*GiST and GIN Indexes for Text Search*) to speed up text searches:

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', body));
```

Notice that the two-argument version of `to_tsvector` is used. Only text search functions that specify a configuration name can be used in expression indexes. This is because the index contents must be unaffected by *default_text_search_config*. If they were affected, the index contents might be inconsistent because different entries could contain `tsvectors` that were created with different text search configurations, and there would be no way to guess which was which. It would be impossible to dump and restore such an index correctly.

Because the two-argument version of `to_tsvector` was used in the index above, only a query reference that uses the two-argument version of `to_tsvector` with the same configuration name will use that index. That is, `WHERE to_tsvector('english', body) @@ 'a & b'` can use the index, but `WHERE to_tsvector(body) @@ 'a & b'` cannot. This ensures that an index will be used only with the same configuration used to create the index entries.

It is possible to set up more complex expression indexes wherein the configuration name is specified by another column, e.g.:

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector(config_name, body));
```

where `config_name` is a column in the `pgweb` table. This allows mixed configurations in the same index while recording which configuration was used for each index entry. This would be useful, for example, if the document collection contained documents in different languages. Again, queries that are meant to use the index must be phrased to match, e.g., `WHERE to_tsvector(config_name, body) @@ 'a & b'`.

Indexes can even concatenate columns:

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', title || '
 ' || body));
```

Another approach is to create a separate `tsvector` column to hold the output of `to_tsvector`. This example is a concatenation of title and body, using `coalesce` to ensure that one field will still be indexed when the other is NULL:

```
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE pgweb SET textsearchable_index_col =
     to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''));
```

Then we create a GIN index to speed up the search:

```
CREATE INDEX textsearch_idx ON pgweb USING gin(textsearchable_index_col);
```

Now we are ready to perform a fast full text search:

```
SELECT title FROM pgweb WHERE textsearchable_index_col @@ to_tsquery('create
 & table')
ORDER BY last_mod_date DESC LIMIT 10;
```

One advantage of the separate-column approach over an expression index is that it is not necessary to explicitly specify the text search configuration in queries in order to make use of the index. As shown in the example above, the query can depend on `default_text_search_config`. Another advantage is that searches will be faster, since it will not be necessary to redo the `to_tsvector` calls to verify index matches. (This is more important when using a GiST index than a GIN index; see *GiST and GIN Indexes for Text Search* .) The expression-index approach is simpler to set up, however, and it requires less disk space since the `tsvector` representation is not stored explicitly.

Controlling Text Search

This topic shows how to create search and query vectors, how to rank search results, and how to highlight search terms in the results of text search queries.

To implement full text searching there must be a function to create a `tsvector` from a document and a `tsquery` from a user query. Also, we need to return results in a useful order, so we need a function that compares documents with respect to their relevance to the query. It's also important to be able to display the results nicely. Greenplum Database provides support for all of these functions.

This topic contains the following subtopics:

- *Parsing Documents*
- *Parsing Queries*
- *Ranking Search Results*
- *Highlighting Results*

Parsing Documents

Greenplum Database provides the function `to_tsvector` for converting a document to the `tsvector` data type.

```
to_tsvector([config regconfig, ] document text) returns tsvector
```

`to_tsvector` parses a textual document into tokens, reduces the tokens to lexemes, and returns a `tsvector` which lists the lexemes together with their positions in the document. The document is processed according to the specified or default text search configuration. Here is a simple example:

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat
 rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

In the example above we see that the resulting tsvector does not contain the words `a`, `on`, or `it`, the word `rats` became `rat`, and the punctuation sign `-` was ignored.

The `to_tsvector` function internally calls a parser which breaks the document text into tokens and assigns a type to each token. For each token, a list of dictionaries (*Text Search Dictionaries*) is consulted, where the list can vary depending on the token type. The first dictionary that *recognizes* the token emits one or more normalized *lexemes* to represent the token. For example, `rats` became `rat` because one of the dictionaries recognized that the word `rats` is a plural form of `rat`. Some words are recognized as *stop words*, which causes them to be ignored since they occur too frequently to be useful in searching. In our example these are `a`, `on`, and `it`. If no dictionary in the list recognizes the token then it is also ignored. In this example that happened to the punctuation sign `-` because there are in fact no dictionaries assigned for its token type (`Space symbols`), meaning space tokens will never be indexed. The choices of parser, dictionaries and which types of tokens to index are determined by the selected text search configuration (*Text Search Configuration Example*). It is possible to have many different configurations in the same database, and predefined configurations are available for various languages. In our example we used the default configuration `english` for the English language.

The function `setweight` can be used to label the entries of a `tsvector` with a given *weight*, where a weight is one of the letters `A`, `B`, `C`, or `D`. This is typically used to mark entries coming from different parts of a document, such as `title` versus `body`. Later, this information can be used for ranking of search results.

Because `to_tsvector(NULL)` will return `NULL`, it is recommended to use `coalesce` whenever a field might be null. Here is the recommended method for creating a tsvector from a structured document:

```
UPDATE tt SET ti = setweight(to_tsvector(coalesce(title,'')), 'A')
  || setweight(to_tsvector(coalesce(keyword,'')), 'B')
  || setweight(to_tsvector(coalesce(abstract,'')), 'C')
  || setweight(to_tsvector(coalesce(body,'')), 'D');
```

Here we have used `setweight` to label the source of each lexeme in the finished `tsvector`, and then merged the labeled `tsvector` values using the `tsvector` concatenation operator `||`. (*Additional Text Search Features* gives details about these operations.)

Parsing Queries

Greenplum Database provides the functions `to_tsquery` and `plainto_tsquery` for converting a query to the `tsquery` data type. `to_tsquery` offers access to more features than `plainto_tsquery`, but is less forgiving about its input.

```
to_tsquery([config regconfig, ] querytext text) returns tsquery
```

`to_tsquery` creates a `tsquery` value from *querytext*, which must consist of single tokens separated by the Boolean operators `&` (AND), `|` (OR), and `!`(NOT). These operators can be grouped using parentheses. In other words, the input to `to_tsquery` must already follow the general rules for `tsquery` input, as described in *Text Search Data Types*. The difference is that while basic `tsquery` input takes the tokens at face value, `to_tsquery` normalizes each token to a lexeme using the specified or default configuration, and discards any tokens that are stop words according to the configuration. For example:

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

As in basic `tsquery` input, weight(s) can be attached to each lexeme to restrict it to match only `tsvector` lexemes of those weight(s). For example:

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
```

```
 'fat' | 'rat':AB
```

Also, * can be attached to a lexeme to specify prefix matching:

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
--------------------------
 'supern':*A & 'star':*AB
```

Such a lexeme will match any word in a `tsvector` that begins with the given string.

`to_tsquery` can also accept single-quoted phrases. This is primarily useful when the configuration includes a thesaurus dictionary that may trigger on such phrases. In the example below, a thesaurus contains the rule `supernovae stars : sn`:

```
SELECT to_tsquery('''supernovae stars'' & !crab');
  to_tsquery
---------------
 'sn' & !'crab'
```

Without quotes, `to_tsquery` will generate a syntax error for tokens that are not separated by an AND or OR operator.

```
plainto_tsquery([ config regconfig, ] querytext ext) returns tsquery
```

`plainto_tsquery` transforms unformatted text *querytext* to `tsquery`. The text is parsed and normalized much as for `to_tsvector`, then the & (AND) Boolean operator is inserted between surviving words.

Example:

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

Note that `plainto_tsquery` cannot recognize Boolean operators, weight labels, or prefix-match labels in its input:

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

Here, all the input punctuation was discarded as being space symbols.

Ranking Search Results

Ranking attempts to measure how relevant documents are to a particular query, so that when there are many matches the most relevant ones can be shown first. Greenplum Database provides two predefined ranking functions, which take into account lexical, proximity, and structural information; that is, they consider how often the query terms appear in the document, how close together the terms are in the document, and how important is the part of the document where they occur. However, the concept of relevancy is vague and very application-specific. Different applications might require additional information for ranking, e.g., document modification time. The built-in ranking functions are only examples. You can write your own ranking functions and/or combine their results with additional factors to fit your specific needs.

The two ranking functions currently available are:

ts_rank([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer]) returns float4	Ranks vectors based on the frequency of their matching lexemes.
ts_rank_cd([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer]) returns float4	This function computes the *cover density* ranking for the given document vector and query, as described in Clarke, Cormack, and Tudhope's "Relevance Ranking for One to Three Term Queries" in the journal "Information Processing and Management", 1999. Cover density is similar to `ts_rank` ranking except that the proximity of matching lexemes to each other is taken into consideration. This function requires lexeme positional information to perform its calculation. Therefore, it ignores any "stripped" lexemes in the `tsvector`. If there are no unstripped lexemes in the input, the result will be zero. (See *Manipulating Documents* for more information about the `strip` function and positional information in `tsvectors`.)

For both these functions, the optional *`weights`* argument offers the ability to weigh word instances more or less heavily depending on how they are labeled. The weight arrays specify how heavily to weigh each category of word, in the order:

```
{D-weight, C-weight, B-weight, A-weight}
```

If no *`weights`* are provided, then these defaults are used:

```
{0.1, 0.2, 0.4, 1.0}
```

Typically weights are used to mark words from special areas of the document, like the title or an initial abstract, so they can be treated with more or less importance than words in the document body.

Since a longer document has a greater chance of containing a query term it is reasonable to take into account document size, e.g., a hundred-word document with five instances of a search word is probably more relevant than a thousand-word document with five instances. Both ranking functions take an integer *`normalization`* option that specifies whether and how a document's length should impact its rank. The integer option controls several behaviors, so it is a bit mask: you can specify one or more behaviors using `|` (for example, `2|4`).

- 0 (the default) ignores the document length
- 1 divides the rank by 1 + the logarithm of the document length
- 2 divides the rank by the document length
- 4 divides the rank by the mean harmonic distance between extents (this is implemented only by `ts_rank_cd`)
- 8 divides the rank by the number of unique words in document
- 16 divides the rank by 1 + the logarithm of the number of unique words in document
- 32 divides the rank by itself + 1

If more than one flag bit is specified, the transformations are applied in the order listed.

It is important to note that the ranking functions do not use any global information, so it is impossible to produce a fair normalization to 1% or 100% as sometimes desired. Normalization option `32 (rank/(rank+1))` can be applied to scale all ranks into the range zero to one, but of course this is just a cosmetic change; it will not affect the ordering of the search results.

Here is an example that selects only the ten highest-ranked matches:

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |   rank
-----------------------------------------------+----------
 Neutrinos in the Sun                          |      3.1
 The Sudbury Neutrino Detector                 |      2.4
 A MACHO View of Galactic Dark Matter          |  2.01317
 Hot Gas and Dark Matter                       |  1.91171
 The Virgo Cluster: Hot Plasma and Dark Matter |  1.90953
 Rafting for Solar Neutrinos                   |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter     |  1.85774
 Hot Gas and Dark Matter                       |   1.6123
 Ice Fishing for Cosmic Neutrinos              |      1.6
 Weak Lensing Distorts the Universe            | 0.818218
```

This is the same example using normalized ranking:

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |        rank
-----------------------------------------------+-------------------
 Neutrinos in the Sun                          | 0.756097569485493
 The Sudbury Neutrino Detector                 | 0.705882361190954
 A MACHO View of Galactic Dark Matter          | 0.668123210574724
 Hot Gas and Dark Matter                       |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
 Rafting for Solar Neutrinos                   | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter     | 0.650072921219637
 Hot Gas and Dark Matter                       | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos              | 0.615384618911517
 Weak Lensing Distorts the Universe            | 0.450010798361481
```

Ranking can be expensive since it requires consulting the tsvector of each matching document, which can be I/O bound and therefore slow. Unfortunately, it is almost impossible to avoid since practical queries often result in large numbers of matches.

Highlighting Results

To present search results it is ideal to show a part of each document and how it is related to the query. Usually, search engines show fragments of the document with marked search terms. Greenplum Database provides a function `ts_headline` that implements this functionality.

```
ts_headline([config regconfig, ] document text, query tsquery [, options
 text ]) returns text
```

`ts_headline` accepts a document along with a query, and returns an excerpt from the document in which terms from the query are highlighted. The configuration to be used to parse the document can be specified by *`config`*; if *`config`* is omitted, the `default_text_search_config` configuration is used.

If an *`options`* string is specified it must consist of a comma-separated list of one or more *`option=value`* pairs. The available options are:

- `StartSel`, `StopSel`: the strings with which to delimit query words appearing in the document, to distinguish them from other excerpted words. You must double-quote these strings if they contain spaces or commas.
- `MaxWords`, `MinWords`: these numbers determine the longest and shortest headlines to output.
- `ShortWord`: words of this length or less will be dropped at the start and end of a headline. The default value of three eliminates common English articles.
- `HighlightAll`: Boolean flag; if `true` the whole document will be used as the headline, ignoring the preceding three parameters.
- `MaxFragments`: maximum number of text excerpts or fragments to display. The default value of zero selects a non-fragment-oriented headline generation method. A value greater than zero selects fragment-based headline generation. This method finds text fragments with as many query words as possible and stretches those fragments around the query words. As a result query words are close to the middle of each fragment and have words on each side. Each fragment will be of at most `MaxWords` and words of length `ShortWord` or less are dropped at the start and end of each fragment. If not all query words are found in the document, then a single fragment of the first `MinWords` in the document will be displayed.
- `FragmentDelimiter`: When more than one fragment is displayed, the fragments will be separated by this string.

Any unspecified options receive these defaults:

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,
MaxFragments=0, FragmentDelimiter=" ... "
```

For example:

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'));
                        ts_headline
------------------------------------------------------------
 containing given <b>query</b> terms
 and return them in order of their <b>similarity</b> to the
 <b>query</b>.

SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'),
  'StartSel = <, StopSel = >');
                       ts_headline
-------------------------------------------------------
 containing given <query> terms
 and return them in order of their <similarity> to the
 <query>.
```

`ts_headline` uses the original document, not a `tsvector` summary, so it can be slow and should be used with care. A typical mistake is to call `ts_headline` for *every* matching document when only ten documents are to be shown. SQL subqueries can help; here is an example:

```
SELECT id, ts_headline(body, q), rank
FROM (SELECT id, body, q, ts_rank_cd(ti, q) AS rank
      FROM apod, to_tsquery('stars') q
      WHERE ti @@ q
```

```
      ORDER BY rank DESC
      LIMIT 10) AS foo;
```

Additional Text Search Features

Greenplum Database has additional functions and operators you can use to manipulate search and query vectors, and to rewite search queries.

This section contains the following subtopics:

- *Manipulating Documents*
- *Manipulating Queries*
- *Rewriting Queries*
- *Gathering Document Statistics*

Manipulating Documents

Parsing Documents showed how raw textual documents can be converted into `tsvector` values. Greenplum Database also provides functions and operators that can be used to manipulate documents that are already in `tsvector` form.

`tsvector || tsvector`	The `tsvector` concatenation operator returns a vector which combines the lexemes and positional information of the two vectors given as arguments. Positions and weight labels are retained during the concatenation. Positions appearing in the right-hand vector are offset by the largest position mentioned in the left-hand vector, so that the result is nearly equivalent to the result of performing `to_tsvector` on the concatenation of the two original document strings. (The equivalence is not exact, because any stop-words removed from the end of the left-hand argument will not affect the result, whereas they would have affected the positions of the lexemes in the right-hand argument if textual concatenation were used.) One advantage of using concatenation in the vector form, rather than concatenating text before applying `to_tsvector`, is that you can use different configurations to parse different sections of the document. Also, because the `setweight` function marks all lexemes of the given vector the same way, it is necessary to parse the text and do `setweight` before concatenating if you want to label different parts of the document with different weights.
`setweight(`*`vector`* `tsvector, `*`weight`* `"char") returns tsvector`	`setweight` returns a copy of the input vector in which every position has been labeled with the given *`weight`*, either `A`, `B`, `C`, or `D`. (`D` is the default for new vectors and as such is not displayed on output.) These labels are retained when vectors are concatenated, allowing words from different parts of a document to be weighted differently by ranking functions.

	Note that weight labels apply to **positions**, not **lexemes**. If the input vector has been stripped of positions then `setweight` does nothing.
length(*vector* tsvector) returns integer	Returns the number of lexemes stored in the vector.
strip(vector tsvector) returns tsvector	Returns a vector which lists the same lexemes as the given vector, but which lacks any position or weight information. While the returned vector is much less useful than an unstripped vector for relevance ranking, it will usually be much smaller.

Manipulating Queries

Parsing Queries showed how raw textual queries can be converted into `tsquery` values. Greenplum Database also provides functions and operators that can be used to manipulate queries that are already in `tsquery` form.

tsquery && tsquery

Returns the AND-combination of the two given queries.

tsquery || tsquery

Returns the OR-combination of the two given queries.

!! tsquery

Returns the negation (NOT) of the given query.

numnode(*query* tsquery) returns integer

Returns the number of nodes (lexemes plus operators) in a tsquery. This function is useful to determine if the **query** is meaningful (returns > 0), or contains only stop words (returns 0). Examples:

```
SELECT numnode(plainto_tsquery('the
 any'));
NOTICE:  query contains only
 stopword(s) or doesn't contain
 lexeme(s), ignored
 numnode
---------
       0

SELECT numnode('foo &
 bar'::tsquery);
 numnode
---------
       3
```

querytree(*query* tsquery) returns text

Returns the portion of a tsquery that can be used for searching an index. This function is useful for detecting unindexable queries, for example those containing only stop words or only negated terms. For example:

```
SELECT querytree(to_tsquery('!
defined'));
 querytree
-----------
```

Rewriting Queries

The `ts_rewrite` family of functions search a given `tsquery` for occurrences of a target subquery, and replace each occurrence with a substitute subquery. In essence this operation is a `tsquery`-specific version of substring replacement. A target and substitute combination can be thought of as a *query rewrite rule*. A collection of such rewrite rules can be a powerful search aid. For example, you can expand the search using synonyms (e.g., `new york`, `big apple`, `nyc`, `gotham`) or narrow the search to direct the user to some hot topic. There is some overlap in functionality between this feature and thesaurus dictionaries (*Thesaurus Dictionary*). However, you can modify a set of rewrite rules on-the-fly without reindexing, whereas updating a thesaurus requires reindexing to be effective.

`ts_rewrite(`*`query`*` tsquery, `*`target`*` tsquery, `*`substitute`*` tsquery) returns tsquery`

This form of `ts_rewrite` simply applies a single rewrite rule: *`target`* is replaced by *`substitute`* wherever it appears in *`query`*. For example:

```
SELECT ts_rewrite('a & b'::tsquery,
 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
------------
 'b' & 'c'
```

`ts_rewrite(`*`query`*` tsquery, `*`select`*` text) returns tsquery`

This form of `ts_rewrite` accepts a starting *`query`* and a SQL *`select`* command, which is given as a text string. The *`select`* must yield two columns of `tsquery` type. For each row of the *`select`* result, occurrences of the first column value (the target) are replaced by the second column value (the substitute) within the current *`query`* value. For example:

```
CREATE TABLE aliases (id int, t
 tsquery, s tsquery);
INSERT INTO aliases VALUES('a',
 'c');

SELECT ts_rewrite('a & b'::tsquery,
 'SELECT t,s FROM aliases');
 ts_rewrite
------------
 'b' & 'c'
```

Note that when multiple rewrite rules are applied in this way, the order of application can be important; so in practice you will want the source query to `ORDER BY` some ordering key.

Let's consider a real-life astronomical example. We'll expand query `supernovae` using table-driven rewriting rules:

```
CREATE TABLE aliases (id int, t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(1, to_tsquery('supernovae'),
 to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT t, s FROM
 aliases');
           ts_rewrite
--------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

We can change the rewriting rules just by updating the table:

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT t, s FROM
 aliases');
                 ts_rewrite
--------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

Rewriting can be slow when there are many rewriting rules, since it checks every rule for a possible match. To filter out obvious non-candidate rules we can use the containment operators for the `tsquery` type. In the example below, we select only those rules which might match the original query:

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b''::tsquery @> t');
 ts_rewrite
------------
 'b' & 'c'
```

Gathering Document Statistics

The function `ts_stat` is useful for checking your configuration and for finding stop-word candidates.

```
ts_stat(sqlquery text, [ weights text, ]
        OUT word text, OUT ndoc integer,
        OUT nentry integer) returns setof record
```

`sqlquery` is a text value containing an SQL query which must return a single `tsvector` column. `ts_stat` executes the query and returns statistics about each distinct lexeme (word) contained in the `tsvector` data. The columns returned are

- *`word`* `text` — the value of a lexeme
- *`ndoc`* `integer` — number of documents (`tsvectors`) the word occurred in
- *`nentry`* `integer` — total number of occurrences of the word

If weights is supplied, only occurrences having one of those weights are counted.

For example, to find the ten most frequent words in a document collection:

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

The same, but counting only word occurrences with weight `A` or `B`:

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

Text Search Parsers

This topic describes the types of tokens the Greenplum Database text search parser produces from raw text.

Text search parsers are responsible for splitting raw document text into *tokens* and identifying each token's type, where the set of possible types is defined by the parser itself. Note that a parser does not modify the text at all — it simply identifies plausible word boundaries. Because of this limited scope,

there is less need for application-specific custom parsers than there is for custom dictionaries. At present Greenplum Database provides just one built-in parser, which has been found to be useful for a wide range of applications.

The built-in parser is named `pg_catalog.default`. It recognizes 23 token types, shown in the following table.

Table 56: Default Parser's Token Types

Alias	Description	Example
asciiword	Word, all ASCII letters	elephant
word	Word, all letters	mañana
numword	Word, letters and digits	beta1
asciihword	Hyphenated word, all ASCII	up-to-date
hword	Hyphenated word, all letters	lógico-matemática
numhword	Hyphenated word, letters and digits	postgresql-beta1
hword_asciipart	Hyphenated word part, all ASCII	postgresql in the context postgresql-beta1
hword_part	Hyphenated word part, all letters	lógico or matemática in the context lógico-matemática
hword_numpart	Hyphenated word part, letters and digits	beta1 in the context postgresql-beta1
email	Email address	foo@example.com
protocol	Protocol head	http://
url	URL	example.com/stuff/index.html
host	Host	example.com
url_path	URL path	/stuff/index.html, in the context of a URL
file	File or path name	/usr/local/foo.txt, if not within a URL
sfloat	Scientific notation	-1.234e56
float	Decimal notation	-1.234
int	Signed integer	-1234
uint	Unsigned integer	1234
version	Version number	8.3.0
tag	XML tag	<a href="dictionaries.html">
entity	XML entity	&
blank	Space symbols	(any whitespace or punctuation not otherwise recognized)

Note:

The parser's notion of a "letter" is determined by the database's locale setting, specifically `lc_ctype`. Words containing only the basic ASCII letters are reported as a separate token type, since it is sometimes useful to distinguish them. In most European languages, token types `word` and `asciiword` should be treated alike.

`email` does not support all valid email characters as defined by RFC 5322. Specifically, the only non-alphanumeric characters supported for email user names are period, dash, and underscore.

It is possible for the parser to produce overlapping tokens from the same piece of text. As an example, a hyphenated word will be reported both as the entire word and as each component:

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |               description                |     token
-----------------+------------------------------------------+---------------
 numhword        | Hyphenated word, letters and digits      | foo-bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank           | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

This behavior is desirable since it allows searches to work for both the whole compound word and for components. Here is another instructive example:

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/
index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

Text Search Dictionaries

Tokens produced by the Greenplum Database full text search parser are passed through a chain of dictionaries to produce a normalized term or "lexeme". Different kinds of dictionaries are available to filter and transform tokens in different ways and for different languages.

This section contains the following subtopics:

- *About Text Search Dictionaries*
- *Stop Words*
- *Simple Dictionary*
- *Synonym Dictionary*
- *Thesaurus Dictionary*
- *Ispell Dictionary*
- *SnowBall Dictionary*

About Text Search Dictionaries

Dictionaries are used to eliminate words that should not be considered in a search (*stop words*), and to *normalize* words so that different derived forms of the same word will match. A successfully normalized word is called a *lexeme*. Aside from improving search quality, normalization and removal of stop words reduces the size of the `tsvector` representation of a document, thereby improving performance. Normalization does not always have linguistic meaning and usually depends on application semantics.

Some examples of normalization:

- Linguistic - Ispell dictionaries try to reduce input words to a normalized form; stemmer dictionaries remove word endings
- URL locations can be canonicalized to make equivalent URLs match:
  - `http://www.pgsql.ru/db/mw/index.html`
  - `http://www.pgsql.ru/db/mw/`
  - `http://www.pgsql.ru/db/../db/mw/index.html`
- Color names can be replaced by their hexadecimal values, e.g., `red`, `green`, `blue`, `magenta` -> `FF0000`, `00FF00`, `0000FF`, `FF00FF`
- If indexing numbers, we can remove some fractional digits to reduce the range of possible numbers, so for example 3.14159265359, 3.1415926, 3.14 will be the same after normalization if only two digits are kept after the decimal point.

A dictionary is a program that accepts a token as input and returns:

- an array of lexemes if the input token is known to the dictionary (notice that one token can produce more than one lexeme)
- a single lexeme with the `TSL_FILTER` flag set, to replace the original token with a new token to be passed to subsequent dictionaries (a dictionary that does this is called a *filtering dictionary*)
- an empty array if the dictionary knows the token, but it is a stop word
- `NULL` if the dictionary does not recognize the input token

Greenplum Database provides predefined dictionaries for many languages. There are also several predefined templates that can be used to create new dictionaries with custom parameters. Each predefined dictionary template is described below. If no existing template is suitable, it is possible to create new ones; see the `contrib/` area of the Greenplum Database distribution for examples.

A text search configuration binds a parser together with a set of dictionaries to process the parser's output tokens. For each token type that the parser can return, a separate list of dictionaries is specified by the configuration. When a token of that type is found by the parser, each dictionary in the list is consulted in turn, until some dictionary recognizes it as a known word. If it is identified as a stop word, or if no dictionary recognizes the token, it will be discarded and not indexed or searched for. Normally, the first dictionary that returns a `non-NULL` output determines the result, and any remaining dictionaries are not consulted; but a filtering dictionary can replace the given word with a modified word, which is then passed to subsequent dictionaries.

The general rule for configuring a list of dictionaries is to place first the most narrow, most specific dictionary, then the more general dictionaries, finishing with a very general dictionary, like a Snowball stemmer or `simple`, which recognizes everything. For example, for an astronomy-specific search (`astro_en` configuration) one could bind token type `asciiword` (ASCII word) to a synonym dictionary of astronomical terms, a general English dictionary and a Snowball English stemmer:

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

A filtering dictionary can be placed anywhere in the list, except at the end where it'd be useless. Filtering dictionaries are useful to partially normalize words to simplify the task of later dictionaries. For example, a filtering dictionary could be used to remove accents from accented letters, as is done by the *unaccent* module.

Stop Words

Stop words are words that are very common, appear in almost every document, and have no discrimination value. Therefore, they can be ignored in the context of full text searching. For example, every English text contains words like a and the, so it is useless to store them in an index. However, stop words do affect the positions in `tsvector`, which in turn affect ranking:

```
SELECT to_tsvector('english','in the list of stop words');
        to_tsvector
```

```
----------------------------
 'list':3 'stop':5 'word':6
```

The missing positions 1,2,4 are because of stop words. Ranks calculated for documents with and without stop words are quite different:

```
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'),
 to_tsquery('list & stop'));
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english','list stop words'),
 to_tsquery('list & stop'));
 ts_rank_cd
------------
        0.1
```

It is up to the specific dictionary how it treats stop words. For example, `ispell` dictionaries first normalize words and then look at the list of stop words, while `Snowball` stemmers first check the list of stop words. The reason for the different behavior is an attempt to decrease noise.

Simple Dictionary

The simple dictionary template operates by converting the input token to lower case and checking it against a file of stop words. If it is found in the file then an empty array is returned, causing the token to be discarded. If not, the lower-cased form of the word is returned as the normalized lexeme. Alternatively, the dictionary can be configured to report non-stop-words as unrecognized, allowing them to be passed on to the next dictionary in the list.

Here is an example of a dictionary definition using the simple template:

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

Here, `english` is the base name of a file of stop words. The file's full name will be `$SHAREDIR/tsearch_data/english.stop`, where `$SHAREDIR` means the Greenplum Database installation's shared-data directory, often `/usr/local/greenplum-db-<version>/share/postgresql` (use `pg_config --sharedir` to determine it if you're not sure). The file format is simply a list of words, one per line. Blank lines and trailing spaces are ignored, and upper case is folded to lower case, but no other processing is done on the file contents.

Now we can test our dictionary:

```
SELECT ts_lexize('public.simple_dict','YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
```

We can also choose to return `NULL`, instead of the lower-cased word, if it is not found in the stop words file. This behavior is selected by setting the dictionary's `Accept` parameter to false. Continuing the example:

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );
```

```
SELECT ts_lexize('public.simple_dict','YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
```

With the default setting of `Accept = true`, it is only useful to place a simple dictionary at the end of a list of dictionaries, since it will never pass on any token to a following dictionary. Conversely, `Accept = false` is only useful when there is at least one following dictionary.

> **Caution:** Most types of dictionaries rely on configuration files, such as files of stop words. These files must be stored in UTF-8 encoding. They will be translated to the actual database encoding, if that is different, when they are read into the server.

> **Caution:** Normally, a database session will read a dictionary configuration file only once, when it is first used within the session. If you modify a configuration file and want to force existing sessions to pick up the new contents, issue an `ALTER TEXT SEARCH DICTIONARY` command on the dictionary. This can be a "dummy" update that doesn't actually change any parameter values.

Synonym Dictionary

This dictionary template is used to create dictionaries that replace a word with a synonym. Phrases are not supported—use the thesaurus template (*Thesaurus Dictionary*) for that. A synonym dictionary can be used to overcome linguistic problems, for example, to prevent an English stemmer dictionary from reducing the word "Paris" to "pari". It is enough to have a `Paris paris` line in the synonym dictionary and put it before the `english_stem` dictionary. For example:

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |  dictionaries  |  dictionary  |
 lexemes
-----------+-----------------+-------+----------------+--------------
+---------
 asciiword | Word, all ASCII | Paris | {english_stem} | english_stem |
 {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |       dictionaries        |
 dictionary | lexemes
-----------+-----------------+-------+---------------------------
+------------+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} |
 my_synonym | {paris}
```

The only parameter required by the synonym template is `SYNONYMS`, which is the base name of its configuration file — `my_synonyms` in the above example. The file's full name will be `$SHAREDIR/tsearch_data/my_synonyms.syn` (where `$SHAREDIR` means the Greenplum Database installation's shared-data directory). The file format is just one line per word to be substituted, with the word followed by its synonym, separated by white space. Blank lines and trailing spaces are ignored.

The synonym template also has an optional parameter `CaseSensitive`, which defaults to `false`. When `CaseSensitive` is `false`, words in the synonym file are folded to lower case, as are input tokens. When it is `true`, words and tokens are not folded to lower case, but are compared as-is.

An asterisk (*) can be placed at the end of a synonym in the configuration file. This indicates that the synonym is a prefix. The asterisk is ignored when the entry is used in `to_tsvector()`, but when it is used in `to_tsquery()`, the result will be a query item with the prefix match marker (see *Parsing Queries*). For example, suppose we have these entries in `$SHAREDIR/tsearch_data/synonym_sample.syn`:

```
postgres pgsql postgresql pgsql postgre pgsql
gogle googl
indices index*
```

Then we will get these results:

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym,
 synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn','indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH
 syn;
mydb=# SELECT to_tsvector('tst','indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst','indices');
 to_tsquery
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
            tsvector
--------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@
 to_tsquery('tst','indices');
 ?column?
----------
 t
(1 row)
```

Thesaurus Dictionary

A thesaurus dictionary (sometimes abbreviated as TZ) is a collection of words that includes information about the relationships of words and phrases, i.e., broader terms (BT), narrower terms (NT), preferred terms, non-preferred terms, related terms, etc.

Basically a thesaurus dictionary replaces all non-preferred terms by one preferred term and, optionally, preserves the original terms for indexing as well. Greenplum Database's current implementation of the

thesaurus dictionary is an extension of the synonym dictionary with added *phrase* support. A thesaurus dictionary requires a configuration file of the following format:

```
# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...
```

where the colon (`:`) symbol acts as a delimiter between a phrase and its replacement.

A thesaurus dictionary uses a *subdictionary* (which is specified in the dictionary's configuration) to normalize the input text before checking for phrase matches. It is only possible to select one subdictionary. An error is reported if the subdictionary fails to recognize a word. In that case, you should remove the use of the word or teach the subdictionary about it. You can place an asterisk (`*`) at the beginning of an indexed word to skip applying the subdictionary to it, but all sample words **must** be known to the subdictionary.

The thesaurus dictionary chooses the longest match if there are multiple phrases matching the input, and ties are broken by using the last definition.

Specific stop words recognized by the subdictionary cannot be specified; instead use ? to mark the location where any stop word can appear. For example, assuming that `a` and `the` are stop words according to the subdictionary:

```
? one ? two : swsw
```

matches `a one the two` and `the one a two`; both would be replaced by `swsw`.

Since a thesaurus dictionary has the capability to recognize phrases it must remember its state and interact with the parser. A thesaurus dictionary uses these assignments to check if it should handle the next word or stop accumulation. The thesaurus dictionary must be configured carefully. For example, if the thesaurus dictionary is assigned to handle only the `asciiword` token, then a thesaurus dictionary definition like `one 7` will not work since token type `uint` is not assigned to the thesaurus dictionary.

> **Caution:** Thesauruses are used during indexing so any change in the thesaurus dictionary's parameters requires reindexing. For most other dictionary types, small changes such as adding or removing stopwords does not force reindexing.

Thesaurus Configuration

To define a new thesaurus dictionary, use the `thesaurus` template. For example:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

Here:

- `thesaurus_simple` is the new dictionary's name
- `mythesaurus` is the base name of the thesaurus configuration file. (Its full name will be `$SHAREDIR/tsearch_data/mythesaurus.ths`, where `$SHAREDIR` means the installation shared-data directory.)
- `pg_catalog.english_stem` is the subdictionary (here, a Snowball English stemmer) to use for thesaurus normalization. Notice that the subdictionary will have its own configuration (for example, stop words), which is not shown here.

Now it is possible to bind the thesaurus dictionary `thesaurus_simple` to the desired token types in a configuration, for example:

```
ALTER TEXT SEARCH CONFIGURATION russian
```

```
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_simple;
```

Thesaurus Example

Consider a simple astronomical thesaurus `thesaurus_astro`, which contains some astronomical word combinations:

```
supernovae stars : sn
crab nebulae : crab
```

Below we create a dictionary and bind some token types to an astronomical thesaurus and English stemmer:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

Now we can see how it works. `ts_lexize` is not very useful for testing a thesaurus, because it treats its input as a single token. Instead we can use `plainto_tsquery` and `to_tsvector`, which will break their input strings into multiple tokens:

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

In principle, one can use `to_tsquery` if you quote the argument:

```
SELECT to_tsquery('''supernova star''');
 to_tsquery
------------
 'sn'
```

Notice that `supernova star` matches `supernovae stars` in `thesaurus_astro` because we specified the `english_stem` stemmer in the `thesaurus` definition. The stemmer removed the `e` and `s`.

To index the original phrase as well as the substitute, just include it in the right-hand part of the definition:

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
       plainto_tsquery
-----------------------------
 'sn' & 'supernova' & 'star'
```

Ispell Dictionary

The Ispell dictionary template supports *morphological dictionaries*, which can normalize many different linguistic forms of a word into the same lexeme. For example, an English Ispell dictionary can match all

declensions and conjugations of the search term `bank`, e.g., `banking`, `banked`, `banks`, `banks'`, and `bank's`.

The standard Greenplum Database distribution does not include any Ispell configuration files. Dictionaries for a large number of languages are available from *Ispell*. Also, some more modern dictionary file formats are supported — *MySpell* (OO < 2.0.1) and *Hunspell* (OO >= 2.0.2). A large list of dictionaries is available on the *OpenOffice Wiki*.

To create an Ispell dictionary, use the built-in `ispell` template and specify several parameters:

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

Here, `DictFile`, `AffFile`, and `StopWords` specify the base names of the dictionary, affixes, and stop-words files. The stop-words file has the same format explained above for the `simple` dictionary type. The format of the other files is not specified here but is available from the above-mentioned web sites.

Ispell dictionaries usually recognize a limited set of words, so they should be followed by another broader dictionary; for example, a Snowball dictionary, which recognizes everything.

Ispell dictionaries support splitting compound words; a useful feature. Notice that the affix file should specify a special flag using the `compoundwords controlled` statement that marks dictionary words that can participate in compound formation:

```
compoundwords  controlled z
```

Here are some examples for the Norwegian language:

```
SELECT ts_lexize('norwegian_ispell',
 'overbuljongterningpakkmesterassistent');
   {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
   {sjokoladefabrikk,sjokolade,fabrikk}
```

> **Note:**
>
> MySpell does not support compound words. Hunspell has sophisticated support for compound words. At present, Greenplum Database implements only the basic compound word operations of Hunspell.

SnowBall Dictionary

The Snowball dictionary template is based on a project by Martin Porter, inventor of the popular Porter's stemming algorithm for the English language. Snowball now provides stemming algorithms for many languages (see the *Snowball site* for more information). Each algorithm understands how to reduce common variant forms of words to a base, or stem, spelling within its language. A Snowball dictionary requires a language parameter to identify which stemmer to use, and optionally can specify a stopword file name that gives a list of words to eliminate. (Greenplum Database's standard stopword lists are also provided by the Snowball project.) For example, there is a built-in definition equivalent to

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

The stopword file format is the same as already explained.

A Snowball dictionary recognizes everything, whether or not it is able to simplify the word, so it should be placed at the end of the dictionary list. It is useless to have it before any other dictionary because a token will never pass through it to the next dictionary.

Text Search Configuration Example

This topic shows how to create a customized text search configuration to process document and query text.

A text search configuration specifies all options necessary to transform a document into a `tsvector`: the parser to use to break text into tokens, and the dictionaries to use to transform each token into a lexeme. Every call of `to_tsvector` or `to_tsquery` needs a text search configuration to perform its processing. The configuration parameter *default_text_search_config* specifies the name of the default configuration, which is the one used by text search functions if an explicit configuration parameter is omitted. It can be set in `postgresql.conf` using the `gpconfig` command-line utility, or set for an individual session using the `SET` command.

Several predefined text search configurations are available, and you can create custom configurations easily. To facilitate management of text search objects, a set of SQL commands is available, and there are several psql commands that display information about text search objects (*psql Support*).

As an example we will create a configuration `pg`, starting by duplicating the built-in `english` configuration:

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

We will use a PostgreSQL-specific synonym list and store it in `$SHAREDIR/tsearch_data/pg_dict.syn`. The file contents look like:

```
postgres    pg
pgsql       pg
postgresql  pg
```

We define the synonym dictionary like this:

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

Next we register the Ispell dictionary `english_ispell`, which has its own configuration files:

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

Now we can set up the mappings for words in configuration `pg`:

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

We choose not to index or search some token types that the built-in configuration does handle:

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

Now we can test our configuration:

```
SELECT * FROM ts_debug('public.pg', '
PostgreSQL, the highly scalable, SQL compliant, open source object-
relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

The next step is to set the session to use the new configuration, which was created in the `public` schema:

```
=> \dF
   List of text search configurations
 Schema  | Name | Description
---------+------+-------------
 public  | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
---------------------------
 public.pg
```

Testing and Debugging Text Search

This topic introduces the Greenplum Database functions you can use to test and debug a search configuration or the individual parser and dictionaries specified in a configuration.

The behavior of a custom text search configuration can easily become confusing. The functions described in this section are useful for testing text search objects. You can test a complete configuration, or test parsers and dictionaries separately.

This section contains the following subtopics:

- *Configuration Testing*
- *Parser Testing*
- *Dictionary Testing*

Configuration Testing

The function `ts_debug` allows easy testing of a text search configuration.

```
ts_debug([config regconfig, ] document text,
         OUT alias text,
         OUT description text,
         OUT token text,
         OUT dictionaries regdictionary[],
         OUT dictionary regdictionary,
         OUT lexemes text[])
         returns setof record
```

`ts_debug` displays information about every token of *`document`* as produced by the parser and processed by the configured dictionaries. It uses the configuration specified by *`config`*, or `default_text_search_config` if that argument is omitted.

`ts_debug` returns one row for each token identified in the text by the parser. The columns returned are

- *`alias`* `text` — short name of the token type
- *`description`* `text` — description of the token type

- *token* `text` — text of the token
- *dictionaries* `regdictionary[]` — the dictionaries selected by the configuration for this token type
- *dictionary* `regdictionary` — the dictionary that recognized the token, or `NULL` if none did
- *lexemes* `text[]` — the lexeme(s) produced by the dictionary that recognized the token, or `NULL` if none did; an empty array (`{}`) means it was recognized as a stop word

Here is a simple example:

```
SELECT * FROM ts_debug('english','a fat  cat sat on a mat - it ate a fat
 rats');
   alias   |   description   | token |  dictionaries  |  dictionary  |
 lexemes
-----------+-----------------+-------+----------------+--------------
+---------
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
```

For a more extensive demonstration, we first create a `public.english` configuration and Ispell dictionary for the English language:

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY =
 pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;
```

```
SELECT * FROM ts_debug('public.english','The Brightest supernovaes');
   alias   |   description   |    token    |         dictionaries          |
   dictionary   |   lexemes
-----------+-----------------+-------------+-------------------------------
+---------------+------------
```

```
 asciiword | Word, all ASCII | The         | {english_ispell,english_stem} |
 english_ispell | {}
 blank     | Space symbols   |             | {}                            |
           |
 asciiword | Word, all ASCII | Brightest   | {english_ispell,english_stem} |
 english_ispell | {bright}
 blank     | Space symbols   |             | {}                            |
           |
 asciiword | Word, all ASCII | supernovaes | {english_ispell,english_stem} |
 english_stem   | {supernova}
```

In this example, the word `Brightest` was recognized by the parser as an `ASCII` word (alias `asciiword`). For this token type the dictionary list is `english_ispell` and `english_stem`. The word was recognized by `english_ispell`, which reduced it to the noun `bright`. The word `supernovaes` is unknown to the `english_ispell` dictionary so it was passed to the next dictionary, and, fortunately, was recognized (in fact, `english_stem` is a Snowball dictionary which recognizes everything; that is why it was placed at the end of the dictionary list).

The word `The` was recognized by the `english_ispell` dictionary as a stop word (*Stop Words*) and will not be indexed. The spaces are discarded too, since the configuration provides no dictionaries at all for them.

You can reduce the width of the output by explicitly specifying which columns you want to see:

```
SELECT alias, token, dictionary, lexemes FROM ts_debug('public.english','The
 Brightest supernovaes');
  alias    |    token    |   dictionary   |    lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

Parser Testing

The following functions allow direct testing of a text search parser.

```
ts_parse(parser_name text, document text,
         OUT tokid integer, OUT token text) returns setof record
ts_parse(parser_oid oid, document text,
         OUT tokid integer, OUT token text) returns setof record
```

`ts_parse` parses the given document and returns a series of records, one for each token produced by parsing. Each record includes a `tokid` showing the assigned token type and a `token`, which is the text of the token. For example:

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
ts_token_type(parser_name text, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
ts_token_type(parser_oid oid, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
```

`ts_token_type` returns a table which describes each type of token the specified parser can recognize. For each token type, the table gives the integer `tokid` that the parser uses to label a token of that type, the `alias` that names the token type in configuration commands, and a short `description`. For example:

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------+-----------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
```

Dictionary Testing

The `ts_lexize` function facilitates dictionary testing.

```
ts_lexize(dictreg dictionary, token text) returns text[]
```

`ts_lexize` returns an array of lexemes if the input *`token`* is known to the dictionary, or an empty array if the token is known to the dictionary but it is a stop word, or `NULL` if it is an unknown word.

Examples:

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

Note: The `ts_lexize` function expects a single token, not text. Here is a case where this can be confusing:

```
SELECT ts_lexize('thesaurus_astro','supernovae stars') is null;
 ?column?
----------
 t
```

The thesaurus dictionary `thesaurus_astro` does know the phrase `supernovae stars`, but `ts_lexize` fails since it does not parse the input text but treats it as a single token. Use `plainto_tsquery` or `to_tsvector` to test thesaurus dictionaries, for example:

```
SELECT plainto_tsquery('supernovae stars');
 plainto_tsquery
-----------------
 'sn'
```

GiST and GIN Indexes for Text Search

This topic describes and compares the Greenplum Database index types that are used for full text searching.

There are two kinds of indexes that can be used to speed up full text searches. Indexes are not mandatory for full text searching, but in cases where a column is searched on a regular basis, an index is usually desirable.

`CREATE INDEX` *`name`* `ON` *`table`* `USING gist(`*`column`*`);`	Creates a GiST (Generalized Search Tree)-based index. The *`column`* can be of `tsvector` or `tsquery` type.
`CREATE INDEX` *`name`* `ON` *`table`* `USING gin(`*`column`*`);`	Creates a GIN (Generalized Inverted Index)-based index. The *`column`* must be of `tsvector` type.

There are substantial performance differences between the two index types, so it is important to understand their characteristics.

A GiST index is *lossy*, meaning that the index may produce false matches, and it is necessary to check the actual table row to eliminate such false matches. (Greenplum Database does this automatically when needed.) GiST indexes are lossy because each document is represented in the index by a fixed-length signature. The signature is generated by hashing each word into a single bit in an n-bit string, with all these bits OR-ed together to produce an n-bit document signature. When two words hash to the same bit position there will be a false match. If all words in the query have matches (real or false) then the table row must be retrieved to see if the match is correct.

Lossiness causes performance degradation due to unnecessary fetches of table records that turn out to be false matches. Since random access to table records is slow, this limits the usefulness of GiST indexes. The likelihood of false matches depends on several factors, in particular the number of unique words, so using dictionaries to reduce this number is recommended.

GIN indexes are not lossy for standard queries, but their performance depends logarithmically on the number of unique words. (However, GIN indexes store only the words (lexemes) of `tsvector` values, and not their weight labels. Thus a table row recheck is needed when using a query that involves weights.)

In choosing which index type to use, GiST or GIN, consider these performance differences:

- GIN index lookups are about three times faster than GiST
- GIN indexes take about three times longer to build than GiST
- GIN indexes are moderately slower to update than GiST indexes, but about 10 times slower if fast-update support was disabled (see *GIN Fast Update Technique* in the PostgreSQL documentation for details)
- GIN indexes are two-to-three times larger than GiST indexes

As a rule of thumb, GIN indexes are best for static data because lookups are faster. For dynamic data, GiST indexes are faster to update. Specifically, GiST indexes are very good for dynamic data and fast if the number of unique words (lexemes) is under 100,000, while GIN indexes will handle 100,000+ lexemes better but are slower to update.

Note that GIN index build time can often be improved by increasing *maintenance_work_mem*, while GiST index build time is not sensitive to that parameter.

Partitioning of big collections and the proper use of GiST and GIN indexes allows the implementation of very fast searches with online update. Partitioning can be done at the database level using table inheritance, or by distributing documents over servers and collecting search results using *dblink*. The latter is possible because ranking functions use only local information.

psql Support

The psql command-line utility provides a meta-command to display information about Greenplum Database full text search configurations.

Information about text search configuration objects can be obtained in psql using a set of commands:

```
\dF{d,p,t}[+] [PATTERN]
```

An optional + produces more details.

The optional parameter PATTERN can be the name of a text search object, optionally schema-qualified. If PATTERN is omitted then information about all visible objects will be displayed. PATTERN can be a regular expression and can provide **separate** patterns for the schema and object names. The following examples illustrate this:

```
=> \dF *fulltext*
       List of text search configurations
 Schema |  Name        | Description
--------+--------------+-------------
 public | fulltext_cfg |
```

```
=> \dF *.fulltext*
       List of text search configurations
 Schema   |  Name        | Description
----------+--------------------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

The available commands are:

\dF[+] [PATTERN]

List text search configurations (add + for more detail).

```
=> \dF russian
            List of text search
 configurations
   Schema   |  Name   |
 Description
------------+---------
+-----------------------------------
 pg_catalog | russian |
 configuration for russian language

=> \dF+ russian
Text search configuration
 "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
```

```
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | russian_stem
```

\dFd[+] [PATTERN]

List text search dictionaries (add + for more detail).

```
=> \dFd
                              List of
 text search dictionaries
   Schema   |      Name       |
                  Description

------------+-----------------
+-----------------------------------------------
 pg_catalog | danish_stem
 | snowball stemmer for danish
 language
 pg_catalog | dutch_stem      |
 snowball stemmer for dutch language
 pg_catalog | english_stem
 | snowball stemmer for english
 language
 pg_catalog | finnish_stem
 | snowball stemmer for finnish
 language
 pg_catalog | french_stem
 | snowball stemmer for french
 language
 pg_catalog | german_stem
 | snowball stemmer for german
 language
 pg_catalog | hungarian_stem  |
 snowball stemmer for hungarian
 language
 pg_catalog | italian_stem
 | snowball stemmer for italian
 language
 pg_catalog | norwegian_stem  |
 snowball stemmer for norwegian
 language
 pg_catalog | portuguese_stem |
 snowball stemmer for portuguese
 language
 pg_catalog | romanian_stem   |
 snowball stemmer for romanian
 language
 pg_catalog | russian_stem
 | snowball stemmer for russian
 language
```

```
 pg_catalog | simple          |
 simple dictionary: just lower case
 and check for stopword
 pg_catalog | spanish_stem
 | snowball stemmer for spanish
 language
 pg_catalog | swedish_stem
 | snowball stemmer for swedish
 language
 pg_catalog | turkish_stem
 | snowball stemmer for turkish
 language
```

\dFp[+] [PATTERN]

List text search parsers (add + for more detail).

```
=> \dFp
        List of text search parsers
   Schema   |  Name   |
 Description
------------+---------
+---------------------
 pg_catalog | default | default word
 parser
=> \dFp+
    Text search parser
 "pg_catalog.default"
     Method      |    Function    |
 Description
-----------------+----------------
+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
 End parse       | prsd_end       |
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

        Token types for parser
 "pg_catalog.default"
   Token name    |
 Description
-----------------
+-----------------------------------------
 asciihword      | Hyphenated word,
 all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
 file            | File or path name
 float           | Decimal notation
 host            | Host
 hword           | Hyphenated word,
 all letters
 hword_asciipart | Hyphenated word
 part, all ASCII
 hword_numpart   | Hyphenated word
 part, letters and digits
 hword_part      | Hyphenated word
 part, all letters
 int             | Signed integer
 numhword        | Hyphenated word,
 letters and digits
```

```
 numword         | Word, letters and
 digits
 protocol        | Protocol head
 sfloat          | Scientific
 notation
 tag             | XML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
 version         | Version number
 word            | Word, all letters
(23 rows)
```

\dFt[+] [PATTERN]

List text search templates (add + for more detail).

```
=> \dFt
                            List of
 text search templates
   Schema   |   Name    |
             Description

------------+-----------
+---------------------------------------------
 pg_catalog | ispell    | ispell
 dictionary
 pg_catalog | simple    | simple
 dictionary: just lower case and
 check for stopword
 pg_catalog | snowball  | snowball
 stemmer
 pg_catalog | synonym   | synonym
 dictionary: replace word by its
 synonym
 pg_catalog | thesaurus | thesaurus
 dictionary: phrase by phrase
 substitution
```

Limitations

This topic lists limitations and maximums for Greenplum Database full text search objects.

The current limitations of Greenplum Database's text search features are:

- The `tsvector` and `tsquery` types are not supported in the distribution key for a Greenplum Database table
- The length of each lexeme must be less than 2K bytes
- The length of a `tsvector` (lexemes + positions) must be less than 1 megabyte
- The number of lexemes must be less than 2^{64}
- Position values in `tsvector` must be greater than 0 and no more than 16,383
- No more than 256 positions per lexeme
- The number of nodes (lexemes + operators) in a tsquery must be less than 32,768

For comparison, the PostgreSQL 8.1 documentation contained 10,441 unique words, a total of 335,420 words, and the most frequent word "postgresql" was mentioned 6,127 times in 655 documents.

Another example — the PostgreSQL mailing list archives contained 910,989 unique words with 57,491,343 lexemes in 461,020 messages.

Using Greenplum MapReduce

MapReduce is a programming model developed by Google for processing and generating large data sets on an array of commodity servers. Greenplum MapReduce allows programmers who are familiar with the MapReduce model to write map and reduce functions and submit them to the Greenplum Database parallel engine for processing.

You configure a Greenplum MapReduce job via a YAML-formatted configuration file, then pass the file to the Greenplum MapReduce program, `gpmapreduce`, for execution by the Greenplum Database parallel engine. The Greenplum Database system distributes the input data, executes the program across a set of machines, handles machine failures, and manages the required inter-machine communication.

Refer to *gpmapreduce* for details about running the Greenplum MapReduce program.

About the Greenplum MapReduce Configuration File

This section explains some basics of the Greenplum MapReduce configuration file format to help you get started creating your own Greenplum MapReduce configuration files. Greenplum uses the *YAML 1.1* document format and then implements its own schema for defining the various steps of a MapReduce job.

All Greenplum MapReduce configuration files must first declare the version of the YAML specification they are using. After that, three dashes (`---`) denote the start of a document, and three dots (`...`) indicate the end of a document without starting a new one. (A document in this context is equivalent to a MapReduce job.) Comment lines are prefixed with a pound symbol (#). You can declare multiple Greenplum MapReduce documents/jobs in the same file:

```
%YAML 1.1
---
# Begin Document 1
# ...
---
# Begin Document 2
# ...
```

Within a Greenplum MapReduce document, there are three basic types of data structures or *nodes*: *scalars*, *sequences* and *mappings*.

A *scalar* is a basic string of text indented by a space. If you have a scalar input that spans multiple lines, a preceding pipe (|) denotes a *literal* style, where all line breaks are significant. Alternatively, a preceding angle bracket (>) folds a single line break to a space for subsequent lines that have the same indentation level. If a string contains characters that have reserved meaning, the string must be quoted or the special character must be escaped with a backslash (\ ).

```
# Read each new line literally
somekey: |   this value contains two lines
   and each line is read literally
# Treat each new line as a space
anotherkey: >
   this value contains two lines
   but is treated as one continuous line
# This quoted string contains a special character
ThirdKey: "This is a string: not a mapping"
```

A *sequence* is a list with each entry in the list on its own line denoted by a dash and a space (`- `). Alternatively, you can specify an inline sequence as a comma-separated list within square brackets. A sequence provides a set of data and gives it an order. When you load a list into the Greenplum MapReduce program, the order is kept.

```
# list sequence
```

```
- this
- is
- a list
- with
- five scalar values
# inline sequence
[this, is, a list, with, five scalar values]
```

A *mapping* is used to pair up data values with indentifiers called *keys*. Mappings use a colon and space (`: `) for each `key: value` pair, or can also be specified inline as a comma-separated list within curly braces. The *key* is used as an index for retrieving data from a mapping.

```
# a mapping of items
title: War and Peace
author: Leo Tolstoy
date: 1865
# same mapping written inline
{title: War and Peace, author: Leo Tolstoy, date: 1865}
```

Keys are used to associate meta information with each node and specify the expected node type (*scalar*, *sequence* or *mapping*).

The Greenplum MapReduce program processes the nodes of a document in order and uses indentation (spaces) to determine the document hierarchy and the relationships of the nodes to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.

Refer to *`gpmapreduce.yaml`* for detailed information about the Greenplum MapReduce configuration file format and the keys and values supported.

Example Greenplum MapReduce Job

In this example, you create a MapReduce job that processes text documents and reports on the number of occurrences of certain keywords in each document. The documents and keywords are stored in separate Greenplum Database tables that you create as part of the exercise.

This example MapReduce job utilizes the untrusted `plpythonu` language; as such, you must run the job as a user with Greenplum Database administrative privileges.

1. Log in to the Greenplum Database master host as the `gpadmin` administrative user and set up your environment. For example:

   ```
   $ ssh gpadmin@<gpmaster>
   gpadmin@gpmaster$ . /usr/local/greenplum-db/greenplum_path.sh
   ```

2. Create a new database for the MapReduce example: For example:

   ```
   gpadmin@gpmaster$ createdb mapredex_db
   ```

3. Start the `psql` subsystem, connecting to the new database:

   ```
   gpadmin@gpmaster$ psql -d mapredex_db
   ```

4. Register the PL/Python language in the database. For example:

   ```
   mapredex_db=> CREATE EXTENSION plpythonu;
   ```

5. Create the `documents` table and add some data to the table. For example:

   ```
   CREATE TABLE documents (doc_id int, url text, data text);
   INSERT INTO documents VALUES (1, 'http:/url/1', 'this is one document in
    the corpus');
   ```

```
INSERT INTO documents VALUES (2, 'http:/url/2', 'i am the second document
 in the corpus');
INSERT INTO documents VALUES (3, 'http:/url/3', 'being third never really
 bothered me until now');
INSERT INTO documents VALUES (4, 'http:/url/4', 'the document before me is
 the third document');
```

6. Create the `keywords` table and add some data to the table. For example:

```
CREATE TABLE keywords (keyword_id int, keyword text);
INSERT INTO keywords VALUES (1, 'the');
INSERT INTO keywords VALUES (2, 'document');
INSERT INTO keywords VALUES (3, 'me');
INSERT INTO keywords VALUES (4, 'being');
INSERT INTO keywords VALUES (5, 'now');
INSERT INTO keywords VALUES (6, 'corpus');
INSERT INTO keywords VALUES (7, 'is');
INSERT INTO keywords VALUES (8, 'third');
```

7. Construct the MapReduce YAML configuration file. For example, open a file named `mymrjob.yaml` in the editor of your choice and copy/paste the following large text block:

```
# This example MapReduce job processes documents and looks for keywords in
 them.
# It takes two database tables as input:
# - documents (doc_id integer, url text, data text)
# - keywords (keyword_id integer, keyword text)#
# The documents data is searched for occurrences of keywords and returns
 results of
# url, data and keyword (a keyword can be multiple words, such as "high
 performance # computing")
%YAML 1.1
---
VERSION: 1.0.0.2

# Connect to Greenplum Database using this database and role
DATABASE: mapredex_db
USER: gpadmin

# Begin definition section
DEFINE:

 # Declare the input, which selects all columns and rows from the
 # 'documents' and 'keywords' tables.
  - INPUT:
      NAME: doc
      TABLE: documents
  - INPUT:
      NAME: kw
      TABLE: keywords
  # Define the map functions to extract terms from documents and keyword
  # This example simply splits on white space, but it would be possible
  # to make use of a python library like nltk (the natural language
 toolkit)
  # to perform more complex tokenization and word stemming.
  - MAP:
      NAME: doc_map
      LANGUAGE: python
      FUNCTION: |
        i = 0            # the index of a word within the document
        terms = {}# a hash of terms and their indexes within the document

        # Lower-case and split the text string on space
```

```
        for term in data.lower().split():
          i = i + 1# increment i (the index)

        # Check for the term in the terms list:
        # if stem word already exists, append the i value to the array
entry
        # corresponding to the term. This counts multiple occurrences of
the word.
        # If stem word does not exist, add it to the dictionary with
position i.
        # For example:
        # data: "a computer is a machine that manipulates data"
        # "a" [1, 4]
        # "computer" [2]
        # "machine" [3]
        # …
          if term in terms:
            terms[term] += ','+str(i)
          else:
            terms[term] = str(i)

        # Return multiple lines for each document. Each line consists of
        # the doc_id, a term and the positions in the data where the term
appeared.
        # For example:
        #   (doc_id => 100, term => "a", [1,4]
        #   (doc_id => 100, term => "computer", [2]
        #    …
        for term in terms:
          yield([doc_id, term, terms[term]])
      OPTIMIZE: STRICT IMMUTABLE
      PARAMETERS:
        - doc_id integer
        - data text
      RETURNS:
        - doc_id integer
        - term text
        - positions text

 # The map function for keywords is almost identical to the one for
documents
 # but it also counts of the number of terms in the keyword.
 - MAP:
      NAME: kw_map
      LANGUAGE: python
      FUNCTION: |
        i = 0
        terms = {}
        for term in keyword.lower().split():
          i = i + 1
          if term in terms:
            terms[term] += ','+str(i)
          else:
            terms[term] = str(i)

        # output 4 values including i (the total count for term in terms):
        yield([keyword_id, i, term, terms[term]])
      OPTIMIZE: STRICT IMMUTABLE
      PARAMETERS:
        - keyword_id integer
        - keyword text
      RETURNS:
        - keyword_id integer
        - nterms integer
```

```
        - term text
        - positions text

 # A TASK is an object that defines an entire INPUT/MAP/REDUCE stage
 # within a Greenplum MapReduce pipeline. It is like EXECUTION, but it is
 # executed only when called as input to other processing stages.
 # Identify a task called 'doc_prep' which takes in the 'doc' INPUT
defined earlier
 # and runs the 'doc_map' MAP function which returns doc_id, term,
[term_position]
 - TASK:
     NAME: doc_prep
     SOURCE: doc
     MAP: doc_map

 # Identify a task called 'kw_prep' which takes in the 'kw' INPUT defined
earlier
 # and runs the kw_map MAP function which returns kw_id, term,
[term_position]
 - TASK:
     NAME: kw_prep
     SOURCE: kw
     MAP: kw_map

 # One advantage of Greenplum MapReduce is that MapReduce tasks can be
 # used as input to SQL operations and SQL can be used to process a
MapReduce task.
 # This INPUT defines a SQL query that joins the output of the
'doc_prep'
 # TASK to that of the 'kw_prep' TASK. Matching terms are output to the
'candidate'
 # list (any keyword that shares at least one term with the document).
 - INPUT:
     NAME: term_join
     QUERY: |
       SELECT doc.doc_id, kw.keyword_id, kw.term, kw.nterms,
              doc.positions as doc_positions,
              kw.positions as kw_positions
         FROM doc_prep doc INNER JOIN kw_prep kw ON (doc.term = kw.term)

 # In Greenplum MapReduce, a REDUCE function is comprised of one or more
functions.
 # A REDUCE has an initial 'state' variable defined for each grouping
key. that is
 # A TRANSITION function adjusts the state for every value in a key
grouping.
 # If present, an optional CONSOLIDATE function combines multiple
 # 'state' variables. This allows the TRANSITION function to be executed
locally at
 # the segment-level and only redistribute the accumulated 'state' over
 # the network. If present, an optional FINALIZE function can be used to
perform
 # final computation on a state and emit one or more rows of output from
the state.
 #
 # This REDUCE function is called 'term_reducer' with a TRANSITION
function
 # called 'term_transition' and a FINALIZE function called
'term_finalizer'
 - REDUCE:
     NAME: term_reducer
     TRANSITION: term_transition
     FINALIZE: term_finalizer
```

```
 - TRANSITION:
     NAME: term_transition
     LANGUAGE: python
     PARAMETERS:
       - state text
       - term text
       - nterms integer
       - doc_positions text
       - kw_positions text
     FUNCTION: |

       # 'state' has an initial value of '' and is a colon delimited set
       # of keyword positions. keyword positions are comma delimited sets
of
       # integers. For example, '1,3,2:4:'
       # If there is an existing state, split it into the set of keyword
positions
       # otherwise construct a set of 'nterms' keyword positions - all
empty
       if state:
         kw_split = state.split(':')
       else:
         kw_split = []
         for i in range(0,nterms):
           kw_split.append('')

       # 'kw_positions' is a comma delimited field of integers indicating
what
       # position a single term occurs within a given keyword.
       # Splitting based on ',' converts the string into a python list.
       # add doc_positions for the current term
       for kw_p in kw_positions.split(','):
         kw_split[int(kw_p)-1] = doc_positions

       # This section takes each element in the 'kw_split' array and
strings
       # them together placing a ':' in between each element from the
array.
       # For example: for the keyword "computer software computer
hardware",
       # the 'kw_split' array matched up to the document data of
       # "in the business of computer software software engineers"
       # would look like: ['5', '6,7', '5', '']
       # and the outstate would look like: 5:6,7:5:
       outstate = kw_split[0]
       for s in kw_split[1:]:
         outstate = outstate + ':' + s
       return outstate

 - FINALIZE:
     NAME: term_finalizer
     LANGUAGE: python
     RETURNS:
       - count integer
     MODE: MULTI
     FUNCTION: |
       if not state:
         yield 0
       kw_split = state.split(':')

       # This function does the following:
       # 1) Splits 'kw_split' on ':'
       #    for example, 1,5,7:2,8 creates '1,5,7' and '2,8'
```

```
        # 2) For each group of positions in 'kw_split', splits the set on
 ','
        #    to create ['1','5','7'] from Set 0: 1,5,7 and
        #    eventually ['2', '8'] from Set 1: 2,8
        # 3) Checks for empty strings
        # 4) Adjusts the split sets by subtracting the position of the
 set
        #      in the 'kw_split' array
        # ['1','5','7'] - 0 from each element = ['1','5','7']
        # ['2', '8'] - 1 from each element = ['1', '7']
        # 5) Resulting arrays after subtracting the offset in step 4 are
        #    intersected and their overlapping values kept:
        #    ['1','5','7'].intersect['1', '7'] = [1,7]
        # 6) Determines the length of the intersection, which is the
 number of
        # times that an entire keyword (with all its pieces) matches in
 the
        #    document data.
        previous = None
        for i in range(0,len(kw_split)):
          isplit = kw_split[i].split(',')
          if any(map(lambda(x): x == '', isplit)):
            yield 0
          adjusted = set(map(lambda(x): int(x)-i, isplit))
          if (previous):
            previous = adjusted.intersection(previous)
          else:
            previous = adjusted

        # return the final count
        if previous:
          yield len(previous)

   # Define the 'term_match' task which is then executed as part
   # of the 'final_output' query. It takes the INPUT 'term_join' defined
   # earlier and uses the REDUCE function 'term_reducer' defined earlier
  - TASK:
      NAME: term_match
      SOURCE: term_join
      REDUCE: term_reducer
  - INPUT:
      NAME: final_output
      QUERY: |
        SELECT doc.*, kw.*, tm.count
        FROM documents doc, keywords kw, term_match tm
        WHERE doc.doc_id = tm.doc_id
          AND kw.keyword_id = tm.keyword_id
          AND tm.count > 0

# Execute this MapReduce job and send output to STDOUT
EXECUTE:
  - RUN:
      SOURCE: final_output
      TARGET: STDOUT
```

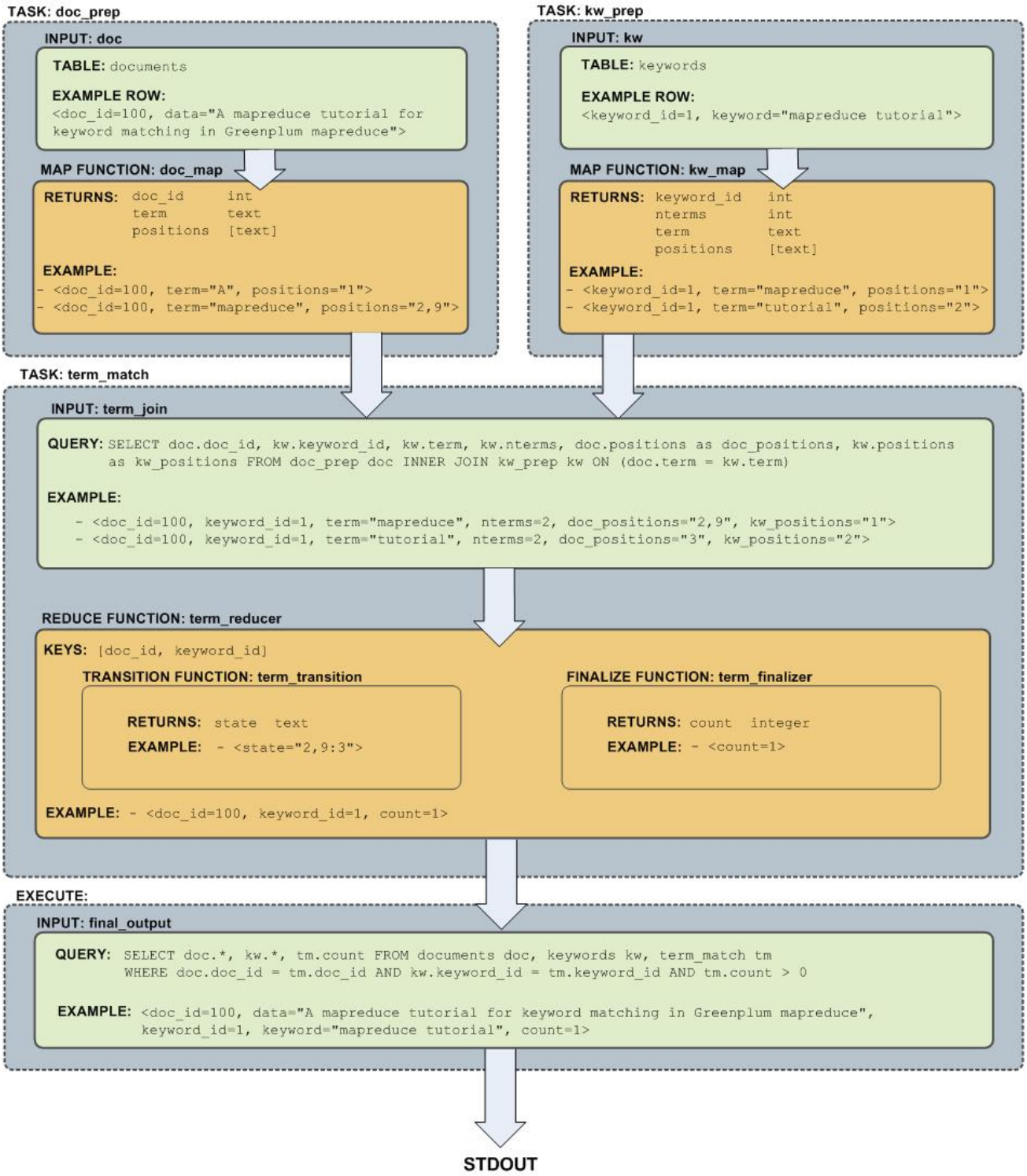
8. Save the file and exit the editor.
9. Run the MapReduce job. For example:

```
gpadmin@gpmaster$ gpmapreduce -f mymrjob.yaml
```

The job displays the number of occurrences of each keyword in each document to `stdout`.

Flow Diagram for MapReduce Example

The following diagram shows the job flow of the MapReduce job defined in the example:



Query Performance

Greenplum Database dynamically eliminates irrelevant partitions in a table and optimally allocates memory for different operators in a query. These enhancements scan less data for a query, accelerate query processing, and support more concurrency.

- Dynamic Partition Elimination

  In Greenplum Database, values available only when a query runs are used to dynamically prune partitions, which improves query processing speed. Enable or disable dynamic partition elimination by setting the server configuration parameter `gp_dynamic_partition_pruning` to `ON` or `OFF`; it is `ON` by default.
- Memory Optimizations

  Greenplum Database allocates memory optimally for different operators in a query and frees and re-allocates memory during the stages of processing a query.

  Note: Greenplum Database uses GPORCA, the Greenplum next generation query optimizer, by default. GPORCA extends the planning and optimization capabilities of the Postgres optimizer. For information about the features and limitations of GPORCA, see *Overview of GPORCA*.

Managing Spill Files Generated by Queries

Greenplum Database creates spill files, also known as workfiles, on disk if it does not have sufficient memory to execute an SQL query in memory.
The maximum number of spill files for a given query is governed by the *gp_workfile_limit_files_per_query* server configuration parameter setting. The default value of 100,000 spill files is sufficient for the majority of queries.

If a query creates more than the configured number of spill files, Greenplum Database returns this error:

```
ERROR: number of workfiles per query limit exceeded
```

Greenplum Database may generate a large number of spill files when:

- Data skew is present in the queried data. To check for data skew, see *Checking for Data Distribution Skew*.
- The amount of memory allocated for the query is too low. You control the maximum amount of memory that can be used by a query with the Greenplum Database server configuration parameters *max_statement_mem* and *statement_mem*, or through resource group or resource queue configuration.

You might be able to run the query successfully by changing the query, changing the data distribution, or changing the system memory configuration. The `gp_toolkit` *gp_workfile_** views display spill file usage information. You can use this information to troubleshoot and tune queries. The *gp_workfile_** views are described in *Checking Query Disk Spill Space Usage*.

Additional documentation resources:

- *Memory Consumption Parameters* identifies the memory-related spill file server configuration parameters.
- *Using Resource Groups* describes memory and spill considerations when resource group-based resource management is active.
- *Using Resource Queues* describes memory and spill considerations when resource queue-based resource management is active.

Query Profiling

Examine the query plans of poorly performing queries to identify possible performance tuning opportunities.

Greenplum Database devises a *query plan* for each query. Choosing the right query plan to match the query and data structure is necessary for good performance. A query plan defines how Greenplum Database will run the query in the parallel execution environment.

The query optimizer uses data statistics maintained by the database to choose a query plan with the lowest possible cost. Cost is measured in disk I/O, shown as units of disk page fetches. The goal is to minimize the total execution cost for the plan.

View the plan for a given query with the `EXPLAIN` command. `EXPLAIN` shows the query optimizer's estimated cost for the query plan. For example:

```
EXPLAIN SELECT * FROM names WHERE id=22;
```

`EXPLAIN ANALYZE` runs the statement in addition to displaying its plan. This is useful for determining how close the optimizer's estimates are to reality. For example:

```
EXPLAIN ANALYZE SELECT * FROM names WHERE id=22;
```

> **Note:** In Greenplum Database, the default GPORCA optimizer co-exists with the Postgres Planner. The `EXPLAIN` output generated by GPORCA is different than the output generated by the Postgres Planner.
>
> By default, Greenplum Database uses GPORCA to generate an execution plan for a query when possible.
>
> When the `EXPLAIN ANALYZE` command uses GPORCA, the `EXPLAIN` plan shows only the number of partitions that are being eliminated. The scanned partitions are not shown. To show name of the scanned partitions in the segment logs set the server configuration parameter `gp_log_dynamic_partition_pruning` to `on`. This example `SET` command enables the parameter.
>
> ```
> SET gp_log_dynamic_partition_pruning = on;
> ```
>
> For information about GPORCA, see *Querying Data*.

Reading EXPLAIN Output

A query plan is a tree of nodes. Each node in the plan represents a single operation, such as a table scan, join, aggregation, or sort.

Read plans from the bottom to the top: each node feeds rows into the node directly above it. The bottom nodes of a plan are usually table scan operations: sequential, index, or bitmap index scans. If the query requires joins, aggregations, sorts, or other operations on the rows, there are additional nodes above the scan nodes to perform these operations. The topmost plan nodes are usually Greenplum Database motion nodes: redistribute, explicit redistribute, broadcast, or gather motions. These operations move rows between segment instances during query processing.

The output of `EXPLAIN` has one line for each node in the plan tree and shows the basic node type and the following execution cost estimates for that plan node:

- **cost** —Measured in units of disk page fetches. 1.0 equals one sequential disk page read. The first estimate is the start-up cost of getting the first row and the second is the total cost of cost of getting all rows. The total cost assumes all rows will be retrieved, which is not always true; for example, if the query uses `LIMIT`, not all rows are retrieved.

  > **Note:** The cost values generated by the Pivotal Query Optimizer and the Postgres Planner are not directly comparable. The two optimizers use different cost models, as well as different algorithms, to determine the cost of an execution plan. Nothing can or should be inferred by comparing cost values between the two optimizers.

In addition, the cost generated for any given optimizer is valid only for comparing plan alternatives for a given single query and set of statistics. Different queries can generate plans with different costs, even when keeping the optimizer a constant.

To summarize, the cost is essentially an internal number used by a given optimizer, and nothing should be inferred by examining only the cost value displayed in the `EXPLAIN` plans.

- **rows** —The total number of rows output by this plan node. This number is usually less than the number of rows processed or scanned by the plan node, reflecting the estimated selectivity of any `WHERE` clause conditions. Ideally, the estimate for the topmost node approximates the number of rows that the query actually returns, updates, or deletes.
- **width** —The total bytes of all the rows that this plan node outputs.

Note the following:

- The cost of a node includes the cost of its child nodes. The topmost plan node has the estimated total execution cost for the plan. This is the number the optimizer intends to minimize.
- The cost reflects only the aspects of plan execution that the query optimizer takes into consideration. For example, the cost does not reflect time spent transmitting result rows to the client.

EXPLAIN Example

The following example describes how to read an `EXPLAIN` query plan for a query:

```
EXPLAIN SELECT * FROM names WHERE name = 'Joelle';
                      QUERY PLAN
-------------------------------------------------------------
Gather Motion 2:1 (slice1) (cost=0.00..20.88 rows=1 width=13)

   -> Seq Scan on 'names' (cost=0.00..20.88 rows=1 width=13)
         Filter: name::text ~~ 'Joelle'::text
```

Read the plan from the bottom to the top. To start, the query optimizer sequentially scans the *names* table. Notice the `WHERE` clause is applied as a *filter* condition. This means the scan operation checks the condition for each row it scans and outputs only the rows that satisfy the condition.

The results of the scan operation are passed to a *gather motion* operation. In Greenplum Database, a gather motion is when segments send rows to the master. In this example, we have two segment instances that send to one master instance. This operation is working on `slice1` of the parallel query execution plan. A query plan is divided into *slices* so the segments can work on portions of the query plan in parallel.

The estimated startup cost for this plan is `00.00` (no cost) and a total cost of `20.88` disk page fetches. The optimizer estimates this query will return one row.

Reading EXPLAIN ANALYZE Output

`EXPLAIN ANALYZE` plans and runs the statement. The `EXPLAIN ANALYZE` plan shows the actual execution cost along with the optimizer's estimates. This allows you to see if the optimizer's estimates are close to reality. `EXPLAIN ANALYZE` also shows the following:

- The total runtime (in milliseconds) in which the query executed.
- The memory used by each slice of the query plan, as well as the memory reserved for the whole query statement.
- The number of *workers* (segments) involved in a plan node operation. Only segments that return rows are counted.
- The maximum number of rows returned by the segment that produced the most rows for the operation. If multiple segments produce an equal number of rows, `EXPLAIN ANALYZE` shows the segment with the longest <*time*> *to end*.
- The segment id of the segment that produced the most rows for an operation.

- For relevant operations, the amount of memory (`work_mem`) used by the operation. If the `work_mem` was insufficient to perform the operation in memory, the plan shows the amount of data spilled to disk for the lowest-performing segment. For example:

  ```
  Work_mem used: 64K bytes avg, 64K bytes max (seg0).
  Work_mem wanted: 90K bytes avg, 90K byes max (seg0) to lessen
  workfile I/O affecting 2 workers.
  ```

- The time (in milliseconds) in which the segment that produced the most rows retrieved the first row, and the time taken for that segment to retrieve all rows. The result may omit *<time> to first row* if it is the same as the *<time> to end*.

EXPLAIN ANALYZE Examples

This example describes how to read an `EXPLAIN ANALYZE` query plan using the same query. The `bold` parts of the plan show actual timing and rows returned for each plan node, as well as memory and time statistics for the whole query.

```
EXPLAIN ANALYZE SELECT * FROM names WHERE name = 'Joelle';
                     QUERY PLAN
-------------------------------------------------------------
Gather Motion 2:1 (slice1; segments: 2) (cost=0.00..20.88 rows=1 width=13)
    Rows out: 1 rows at destination with 0.305 ms to first row, 0.537 ms to
 end, start offset by 0.289 ms.
       -> Seq Scan on names (cost=0.00..20.88 rows=1 width=13)
            Rows out: Avg 1 rows x 2 workers. Max 1 rows (seg0) with 0.255
 ms to first row, 0.486 ms to end, start offset by 0.968 ms.
              Filter: name = 'Joelle'::text
 Slice statistics:

     (slice0) Executor memory: 135K bytes.

   (slice1) Executor memory: 151K bytes avg x 2 workers, 151K bytes max
 (seg0).

Statement statistics:
 Memory used: 128000K bytes
 Total runtime: 22.548 ms
```

Read the plan from the bottom to the top. The total elapsed time to run this query was *22.548* milliseconds.

The *sequential scan* operation had only one segment (*seg0*) that returned rows, and it returned just *1 row*. It took *0.255* milliseconds to find the first row and *0.486* to scan all rows. This result is close to the optimizer's estimate: the query optimizer estimated it would return one row for this query. The *gather motion* (segments sending data to the master) received 1 row . The total elapsed time for this operation was *0.537* milliseconds.

Determining the Query Optimizer

You can view `EXPLAIN` output to determine if GPORCA is enabled for the query plan and whether GPORCA or the Postgres Planner generated the explain plan. The information appears at the end of the `EXPLAIN` output. The `Settings` line displays the setting of the server configuration parameter `OPTIMIZER`. The `Optimizer status` line displays whether GPORCA or the Postgres Planner generated the explain plan.

For these two example query plans, GPORCA is enabled, the server configuration parameter `OPTIMIZER` is `on`. For the first plan, GPORCA generated the `EXPLAIN` plan. For the second plan, Greenplum Database fell back to the Postgres Planner to generate the query plan.

```
                        QUERY PLAN
------------------------------------------------------------------------------
```

```
 Aggregate  (cost=0.00..296.14 rows=1 width=8)
   ->  Gather Motion 2:1  (slice1; segments: 2)  (cost=0.00..295.10 rows=1
 width=8)
         ->  Aggregate  (cost=0.00..294.10 rows=1 width=8)
               ->  Seq Scan on part  (cost=0.00..97.69 rows=100040 width=1)
 Settings:  optimizer=on
 Optimizer status: Pivotal Optimizer (GPORCA) version 1.584
(5 rows)
```

```
explain select count(*) from part;

                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=3519.05..3519.06 rows=1 width=8)
   ->  Gather Motion 2:1  (slice1; segments: 2)  (cost=3518.99..3519.03
 rows=1 width=8)
         ->  Aggregate  (cost=3518.99..3519.00 rows=1 width=8)
               ->  Seq Scan on part  (cost=0.00..3018.79 rows=100040
 width=1)
 Settings:  optimizer=on
 Optimizer status: Postgres query optimizer
(5 rows)
```

For this query, the server configuration parameter `OPTIMIZER` is `off`.

```
explain select count(*) from part;

                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=3519.05..3519.06 rows=1 width=8)
   ->  Gather Motion 2:1  (slice1; segments: 2)  (cost=3518.99..3519.03
 rows=1 width=8)
         ->  Aggregate  (cost=3518.99..3519.00 rows=1 width=8)
               ->  Seq Scan on part  (cost=0.00..3018.79 rows=100040
 width=1)
 Settings: optimizer=off
 Optimizer status: Postgres query optimizer
(5 rows)
```

Examining Query Plans to Solve Problems

If a query performs poorly, examine its query plan and ask the following questions:

- **Do operations in the plan take an exceptionally long time?** Look for an operation consumes the majority of query processing time. For example, if an index scan takes longer than expected, the index could be out-of-date and need to be reindexed. Or, adjust `enable_<operator>` parameters to see if you can force the Postgres Planner to choose a different plan by disabling a particular query plan operator for that query.
- **Does the query planning time exceed query execution time?** When the query involves many table joins, the Postgres Planner uses a dynamic algorithm to plan the query that is in part based on the number of table joins. You can reduce the amount of time that the Postgres Planner spends planning the query by setting the `join_collapse_limit` and `from_collapse_limit` server configuration parameters to a smaller value, such as `8`. Note that while smaller values reduce planning time, they may also yield inferior query plans.
- **Are the optimizer's estimates close to reality?** Run `EXPLAIN ANALYZE` and see if the number of rows the optimizer estimates is close to the number of rows the query operation actually returns. If there is a large discrepancy, collect more statistics on the relevant columns.

  See the *Greenplum Database Reference Guide* for more information on the `EXPLAIN ANALYZE` and `ANALYZE` commands.

- **Are selective predicates applied early in the plan?** Apply the most selective filters early in the plan so fewer rows move up the plan tree. If the query plan does not correctly estimate query predicate selectivity, collect more statistics on the relevant columns. See the `ANALYZE` command in the *Greenplum Database Reference Guide* for more information collecting statistics. You can also try reordering the `WHERE` clause of your SQL statement.
- **Does the optimizer choose the best join order?** When you have a query that joins multiple tables, make sure that the optimizer chooses the most selective join order. Joins that eliminate the largest number of rows should be done earlier in the plan so fewer rows move up the plan tree.

  If the plan is not choosing the optimal join order, set `join_collapse_limit=1` and use explicit `JOIN` syntax in your SQL statement to force the Postgres Planner to the specified join order. You can also collect more statistics on the relevant join columns.

  See the `ANALYZE` command in the *Greenplum Database Reference Guide* for more information collecting statistics.
- **Does the optimizer selectively scan partitioned tables?** If you use table partitioning, is the optimizer selectively scanning only the child tables required to satisfy the query predicates? Scans of the parent tables should return 0 rows since the parent tables do not contain any data. See *Verifying Your Partition Strategy* for an example of a query plan that shows a selective partition scan.
- **Does the optimizer choose hash aggregate and hash join operations where applicable?** Hash operations are typically much faster than other types of joins or aggregations. Row comparison and sorting is done in memory rather than reading/writing from disk. To enable the query optimizer to choose hash operations, there must be sufficient memory available to hold the estimated number of rows. Try increasing work memory to improve performance for a query. If possible, run an `EXPLAIN ANALYZE` for the query to show which plan operations spilled to disk, how much work memory they used, and how much memory was required to avoid spilling to disk. For example:

  ```
  Work_mem used: 23430K bytes avg, 23430K bytes max (seg0). Work_mem wanted:
  33649K bytes avg, 33649K bytes max (seg0) to lessen workfile I/O affecting 2
  workers.
  ```

  The "bytes wanted" message from `EXPLAIN ANALYZE` is based on the amount of data written to work files and is not exact. The minimum `work_mem` needed can differ from the suggested value.

Working with External Data

Both external and foreign tables provide access to data stored in data sources outside of Greenplum Database as if the data were stored in regular database tables. You can read data from and write data to external and foreign tables.

An external table is a Greenplum Database table backed with data that resides outside of the database. You create a readable external table to read data from the external data source and create a writable external table to write data to the external source. You can use external tables in SQL commands just as you would a regular database table. For example, you can `SELECT` (readable external table), `INSERT` (writable external table), and join external tables with other Greenplum tables. External tables are most often used to load and unload database data. Refer to *Defining External Tables* for more information about using external tables to access external data.

Accessing External Data with PXF describes using PXF and external tables to access external data sources.

A foreign table is a different kind of Greenplum Database table backed with data that resides outside of the database. You can both read from and write to the same foreign table. You can similarly use foreign tables in SQL commands as described above for external tables. Refer to *Accessing External Data with Foreign Tables* for more information about accessing external data using foreign tables.

Web-based external tables provide access to data served by an HTTP server or an operating system process. See *Creating and Using External Web Tables* for more about web-based tables.

Accessing External Data with PXF

Data managed by your organization may already reside in external sources such as Hadoop, object stores, and other SQL databases. The Greenplum Platform Extension Framework (PXF) provides access to this external data via built-in connectors that map an external data source to a Greenplum Database table definition.

PXF is installed with Hadoop and Object Storage connectors. These connectors enable you to read external data stored in text, Avro, JSON, RCFile, Parquet, SequenceFile, and ORC formats. You can use the JDBC connector to access an external SQL database.

> **Note:** In previous versions of Greenplum Database, you may have used the `gphdfs` external table protocol to access data stored in Hadoop. Greenplum Database version 6.0.0 removes the `gphdfs` protocol. Use PXF and the `pxf` external table protocol to access Hadoop in Greenplum Database version 6.x.

The Greenplum Platform Extension Framework includes a C-language extension and a Java service. After you configure and initialize PXF, you start a single PXF JVM process on each Greenplum Database segment host. This long- running process concurrently serves multiple query requests.

For detailed information about the architecture of and using PXF, refer to the *Greenplum Platform Extension Framework (PXF)* documentation.

Defining External Tables

External tables enable accessing external data as if it were a regular database table. They are often used to move data into and out of a Greenplum database.

To create an external table definition, you specify the format of your input files and the location of your external data sources. For information about input file formats, see *Formatting Data Files*.

Use one of the following protocols to access external table data sources. You cannot mix protocols in `CREATE EXTERNAL TABLE` statements:

- `file://` accesses external data files on segment hosts that the Greenplum Database superuser (`gpadmin`) can access. See *file:// Protocol*.
- `gpfdist://` points to a directory on the file host and serves external data files to all Greenplum Database segments in parallel. See *gpfdist:// Protocol*.
- `gpfdists://` is the secure version of `gpfdist`. See *gpfdists:// Protocol*.
- The `pxf://` protocol accesses object store systems (Azure, Google Cloud Storage, Minio, S3), external Hadoop systems (HDFS, Hive, HBase), and SQL databases using the Greenplum Platform Extension Framework (PXF). See *pxf:// Protocol*.
- `s3://` accesses files in an Amazon S3 bucket. See *s3:// Protocol*.

The `pxf://` and `s3://` protocols are custom data access protocols, where the `file://`, `gpfdist://`, and `gpfdists://` protocols are implemented internally in Greenplum Database. The custom and internal protocols differ in these ways:

- `pxf://` and `s3://` are custom protocols that must be registered using the `CREATE EXTENSION` command (`pxf`) or the `CREATE PROTOCOL` command (`s3`). Registering the PXF extension in a database creates the `pxf` protocol. (See *Accessing External Data with PXF*.) To use the `s3` protocol, you must configure the database and register the `s3` protocol. (See *Configuring and Using S3 External Tables*.) Internal protocols are always present and cannot be unregistered.
- When a custom protocol is registered, a row is added to the `pg_extprotocol` catalog table to specify the handler functions that implement the protocol. The protocol's shared libraries must have been installed on all Greenplum Database hosts. The internal protocols are not represented in the `pg_extprotocol` table and have no additional libraries to install.
- To grant users permissions on custom protocols, you use `GRANT [SELECT | INSERT | ALL] ON PROTOCOL`. To allow (or deny) users permissions on the internal protocols, you use `CREATE ROLE` or `ALTER ROLE` to add the `CREATEEXTTABLE` (or `NOCREATEEXTTABLE`) attribute to each user's role.

External tables access external files from within the database as if they are regular database tables. External tables defined with the `gpfdist`/`gpfdists`, `pxf`, and `s3` protocols utilize Greenplum parallelism by using the resources of all Greenplum Database segments to load or unload data. The `pxf` protocol leverages the parallel architecture of the Hadoop Distributed File System to access files on that system. The `s3` protocol utilizes the Amazon Web Services (AWS) capabilities.

You can query external table data directly and in parallel using SQL commands such as `SELECT`, `JOIN`, or `SORT EXTERNAL TABLE DATA`, and you can create views for external tables.

The steps for using external tables are:

1. Define the external table.

   To use the `pxf` or `s3` protocol, you must also configure Greenplum Database and enable the protocol. See *pxf:// Protocol* or *s3:// Protocol*.
2. Do one of the following:
   - Start the Greenplum Database file server(s) when using the `gpfdist` or `gpdists` protocols.
   - Verify the configuration for the PXF service and start the service.
   - Verify the Greenplum Database configuration for the `s3` protocol.
3. Place the data files in the correct locations.
4. Query the external table with SQL commands.

Greenplum Database provides readable and writable external tables:

- Readable external tables for data loading. Readable external tables support:
  - Basic extraction, transformation, and loading (ETL) tasks common in data warehousing
  - Reading external table data in parallel from multiple Greenplum database segment instances, to optimize large load operations
  - Filter pushdown. If a query contains a `WHERE` clause, it may be passed to the external data source. Refer to the *gp_external_enable_filter_pushdown* server configuration parameter discussion for

more information. Note that this feature is currently supported only with the `pxf` protocol (see *pxf:// Protocol*).

Readable external tables allow only `SELECT` operations.

- Writable external tables for data unloading. Writable external tables support:
  - Selecting data from database tables to insert into the writable external table
  - Sending data to an application as a stream of data. For example, unload data from Greenplum Database and send it to an application that connects to another database or ETL tool to load the data elsewhere
  - Receiving output from Greenplum parallel MapReduce calculations.

  Writable external tables allow only `INSERT` operations.

External tables can be file-based or web-based. External tables using the `file://` protocol are read-only tables.

- Regular (file-based) external tables access static flat files. Regular external tables are rescannable: the data is static while the query runs.
- Web (web-based) external tables access dynamic data sources, either on a web server with the `http://` protocol or by executing OS commands or scripts. External web tables are not rescannable: the data can change while the query runs.

Greenplum Database backup and restore operations back up and restore only external and external web table *definitions*, not the data source data.

file:// Protocol

The `file://` protocol is used in a URI that specifies the location of an operating system file.

The URI includes the host name, port, and path to the file. Each file must reside on a segment host in a location accessible by the Greenplum Database superuser (`gpadmin`). The host name used in the URI must match a segment host name registered in the `gp_segment_configuration` system catalog table.

The `LOCATION` clause can have multiple URIs, as shown in this example:

```
CREATE EXTERNAL TABLE ext_expenses (
   name text, date date, amount float4, category text, desc1 text )
LOCATION ('file://host1:5432/data/expense/*.csv',
          'file://host2:5432/data/expense/*.csv',
          'file://host3:5432/data/expense/*.csv')
FORMAT 'CSV' (HEADER);
```

The number of URIs you specify in the `LOCATION` clause is the number of segment instances that will work in parallel to access the external table. For each URI, Greenplum assigns a primary segment on the specified host to the file. For maximum parallelism when loading data, divide the data into as many equally sized files as you have primary segments. This ensures that all segments participate in the load. The number of external files per segment host cannot exceed the number of primary segment instances on that host. For example, if your array has four primary segment instances per segment host, you can place four external files on each segment host. Tables based on the `file://` protocol can only be readable tables.

The system view `pg_max_external_files` shows how many external table files are permitted per external table. This view lists the available file slots per segment host when using the `file://` `protocol`. The view is only applicable for the `file:// protocol`. For example:

```
SELECT * FROM pg_max_external_files;
```

gpfdist:// Protocol

The `gpfdist://` protocol is used in a URI to reference a running `gpfdist` instance.

The *`gpfdist`* utility serves external data files from a directory on a file host to all Greenplum Database segments in parallel.

`gpfdist` is located in the `$GPHOME/bin` directory on your Greenplum Database master host and on each segment host.

Run `gpfdist` on the host where the external data files reside. For readable external tables, `gpfdist` uncompresses `gzip` (`.gz`) and `bzip2` (`.bz2`) files automatically. For writable external tables, data is compressed using `gzip` if the target file has a `.gz` extension. You can use the wildcard character (*) or other C-style pattern matching to denote multiple files to read. The files specified are assumed to be relative to the directory that you specified when you started the `gpfdist` instance.

> **Note:** Compression is not supported for readable and writeable external tables when the `gpfdist` utility runs on Windows platforms.

All primary segments access the external file(s) in parallel, subject to the number of segments set in the `gp_external_max_segments` server configuration parameter. Use multiple `gpfdist` data sources in a `CREATE EXTERNAL TABLE` statement to scale the external table's scan performance.

`gpfdist` supports data transformations. You can write a transformation process to convert external data from or to a format that is not directly supported with Greenplum Database external tables.

For more information about configuring `gpfdist`, see *Using the Greenplum Parallel File Server (gpfdist)*.

See the `gpfdist` reference documentation for more information about using `gpfdist` with external tables.

gpfdists:// Protocol

The `gpfdists://` protocol is a secure version of the `gpfdist:// protocol`.

To use it, you run the *`gpfdist`* utility with the `--ssl` option. When specified in a URI, the `gpfdists://` protocol enables encrypted communication and secure identification of the file server and the Greenplum Database to protect against attacks such as eavesdropping and man-in-the-middle attacks.

`gpfdists` implements SSL security in a client/server scheme with the following attributes and limitations:

- Client certificates are required.
- Multilingual certificates are not supported.
- A Certificate Revocation List (CRL) is not supported.
- The `TLSv1` protocol is used with the `TLS_RSA_WITH_AES_128_CBC_SHA` encryption algorithm.
- SSL parameters cannot be changed.
- SSL renegotiation is supported.
- The SSL ignore host mismatch parameter is set to `false`.
- Private keys containing a passphrase are not supported for the `gpfdist` file server (server.key) and for the Greenplum Database (client.key).
- Issuing certificates that are appropriate for the operating system in use is the user's responsibility. Generally, converting certificates as shown in *https://www.sslshopper.com/ssl-converter.html* is supported.

  > **Note:** A server started with the `gpfdist --ssl` option can only communicate with the `gpfdists` protocol. A server that was started with `gpfdist` without the `--ssl` option can only communicate with the `gpfdist` protocol.
- The client certificate file, client.crt
- The client private key file, client.key

Use one of the following methods to invoke the `gpfdists` protocol.

- Run `gpfdist` with the `--ssl` option and then use the `gpfdists` protocol in the `LOCATION` clause of a `CREATE EXTERNAL TABLE` statement.
- Use a `gpload` YAML control file with the `SSL` option set to true. Running `gpload` starts the `gpfdist` server with the `--ssl` option, then uses the `gpfdists` protocol.

Using `gpfdists` requires that the following client certificates reside in the `$PGDATA/gpfdists` directory on each segment.

- The client certificate file, `client.crt`
- The client private key file, `client.key`
- The trusted certificate authorities, `root.crt`

For an example of loading data into an external table security, see *Example 3—Multiple gpfdists instances* .

The server configuration parameter *`verify_gpfdists_cert`* controls whether SSL certificate authentication is enabled when Greenplum Database communicates with the `gpfdist` utility to either read data from or write data to an external data source. You can set the parameter value to `false` to disable authentication when testing the communication between the Greenplum Database external table and the `gpfdist` utility that is serving the external data. If the value is `false`, these SSL exceptions are ignored:

- The self-signed SSL certificate that is used by `gpfdist` is not trusted by Greenplum Database.
- The host name contained in the SSL certificate does not match the host name that is running `gpfdist`.

> **Warning:** Disabling SSL certificate authentication exposes a security risk by not validating the `gpfdists` SSL certificate.

pxf:// Protocol

You can use the Greenplum Platform Extension Framework (PXF) `pxf://` protocol to access data residing in object store systems (Azure, Google Cloud Storage, Minio, S3), external Hadoop systems (HDFS, Hive, HBase), and SQL databases.

The PXF `pxf` protocol is packaged as a Greenplum Database extension. The `pxf` protocol supports reading from external data stores. You can also write text, binary, and parquet-format data with the `pxf` protocol.

When you use the `pxf` protocol to query an external data store, you specify the directory, file, or table that you want to access. PXF requests the data from the data store and delivers the relevant portions in parallel to each Greenplum Database segment instance serving the query.

You must explicitly initialize and start PXF before you can use the `pxf` protocol to read or write external data. You must also enable PXF in each database in which you want to allow users to create external tables to access external data, and grant permissions on the `pxf` protocol to those Greenplum Database users.

For detailed information about configuring and using PXF and the `pxf` protocol, refer to *Accessing External Data with PXF*.

s3:// Protocol

The `s3` protocol is used in a URL that specifies the location of an Amazon S3 bucket and a prefix to use for reading or writing files in the bucket.

Amazon Simple Storage Service (Amazon S3) provides secure, durable, highly-scalable object storage. For information about Amazon S3, see *Amazon S3*.

You can define read-only external tables that use existing data files in the S3 bucket for table data, or writable external tables that store the data from INSERT operations to files in the S3 bucket. Greenplum Database uses the S3 URL and prefix specified in the protocol URL either to select one or more files for a read-only table, or to define the location and filename format to use when uploading S3 files for `INSERT` operations to writable tables.

The s3 protocol also supports *Dell EMC Elastic Cloud Storage* (ECS), an Amazon S3 compatible service.

> **Note:** The pxf protocol can access data in S3 and other object store systems such as Azure, Google Cloud Storage, and Minio. The pxf protocol can also access data in external Hadoop systems (HDFS, Hive, HBase), and SQL databases. See *pxf:// Protocol*.

This topic contains the sections:

Configuring and Using S3 External Tables

Follow these basic steps to configure the S3 protocol and use S3 external tables, using the available links for more information. See also *s3 Protocol Limitations* to better understand the capabilities and limitations of S3 external tables:

1. Configure each database to support the s3 protocol:
   a. In each database that will access an S3 bucket with the s3 protocol, create the read and write functions for the s3 protocol library:

```
CREATE OR REPLACE FUNCTION write_to_s3() RETURNS integer AS
   '$libdir/gps3ext.so', 's3_export' LANGUAGE C STABLE;
```

```
CREATE OR REPLACE FUNCTION read_from_s3() RETURNS integer AS
   '$libdir/gps3ext.so', 's3_import' LANGUAGE C STABLE;
```

   b. In each database that will access an S3 bucket, declare the s3 protocol and specify the read and write functions you created in the previous step:

```
CREATE PROTOCOL s3 (writefunc = write_to_s3, readfunc = read_from_s3);
```

      Note: The protocol name s3 must be the same as the protocol of the URL specified for the external table you create to access an S3 resource.

      The corresponding function is called by every Greenplum Database segment instance. All segment hosts must have access to the S3 bucket.
2. On each Greenplum Database segment, create and install the s3 protocol configuration file:
   a. Create a template s3 protocol configuration file using the gpcheckcloud utility:

```
gpcheckcloud -t > ./mytest_s3.config
```

   b. Edit the template file to specify the accessid and secret required to connect to the S3 location. See *s3 Protocol Configuration File* for information about other s3 protocol configuration parameters.
   c. Copy the file to the same location and filename for all Greenplum Database segments on all hosts. The default file location is *gpseg_data_dir/gpseg_prefixN*/s3/s3.conf. *gpseg_data_dir* is the path to the Greenplum Database segment data directory, *gpseg_prefix* is the segment prefix, and *N* is the segment ID. The segment data directory, prefix, and ID are set when you initialize a Greenplum Database system.

If you copy the file to a different location or filename, then you must specify the location with the `config` parameter in the `s3` protocol URL. See *About the s3 Protocol config Parameter*.

d. Use the `gpcheckcloud` utility to validate connectivity to the S3 bucket:

```
gpcheckcloud -c "s3://<s3-endpoint>/<s3-bucket> config=./
mytest_s3.config"
```

Specify the correct path to the configuration file for your system, as well as the S3 endpoint name and bucket that you want to check. `gpcheckcloud` attempts to connect to the S3 endpoint and lists any files in the S3 bucket, if available. A successful connection ends with the message:

```
Your configuration works well.
```

You can optionally use `gpcheckcloud` to validate uploading to and downloading from the S3 bucket, as described in *Using the gpcheckcloud Utility*.

3. After completing the previous steps to create and configure the `s3` protocol, you can specify an `s3` protocol URL in the `CREATE EXTERNAL TABLE` command to define S3 external tables. For read-only S3 tables, the URL defines the location and prefix used to select existing data files that comprise the S3 table. For example:

```
CREATE READABLE EXTERNAL TABLE S3TBL (date text, time text, amt int)
   LOCATION('s3://s3-us-west-2.amazonaws.com/s3test.example.com/dataset1/
normal/ config=/home/gpadmin/aws_s3/s3.conf')
   FORMAT 'csv';
```

For writable S3 tables, the protocol URL defines the S3 location in which Greenplum database stores data files for the table, as well as a prefix to use when creating files for table `INSERT` operations. For example:

```
CREATE WRITABLE EXTERNAL TABLE S3WRIT (LIKE S3TBL)
   LOCATION('s3://s3-us-west-2.amazonaws.com/s3test.example.com/dataset1/
normal/ config=/home/gpadmin/aws_s3/s3.conf')
   FORMAT 'csv';
```

See *About the S3 Protocol URL* for more information.

About the S3 Protocol URL

For the `s3` protocol, you specify a location for files and an optional configuration file location in the `LOCATION` clause of the `CREATE EXTERNAL TABLE` command. This is the syntax:

```
's3://S3_endpoint[:port]/bucket_name/[S3_prefix] [region=S3_region]
 [config=config_file_location]'
```

The `s3` protocol requires that you specify the S3 endpoint and S3 bucket name. Each Greenplum Database segment instance must have access to the S3 location. The optional *S3_prefix* value is used to select files for read-only S3 tables, or as a filename prefix to use when uploading files for S3 writable tables.

Note: The Greenplum Database `s3` protocol URL must include the S3 endpoint hostname.

To specify an ECS endpoint (an Amazon S3 compatible service) in the `LOCATION` clause, you must set the `s3` configuration file parameter `version` to 2. The `version` parameter controls whether the `region` parameter is used in the `LOCATION` clause. You can also specify an Amazon S3 location when the `version` parameter is 2. For information about `version` parameter, see *s3 Protocol Configuration File*.

> **Note:** Although the *S3_prefix* is an optional part of the syntax, you should always include an S3 prefix for both writable and read-only S3 tables to separate datasets as part of the *CREATE EXTERNAL TABLE* syntax.

For writable S3 tables, the `s3` protocol URL specifies the endpoint and bucket name where Greenplum Database uploads data files for the table. The S3 bucket permissions must be `Upload/Delete` for the S3 user ID that uploads the files. The S3 file prefix is used for each new file uploaded to the S3 location as a result of inserting data to the table. See *About S3 Data Files*.

For read-only S3 tables, the S3 file prefix is optional. If you specify an *S3_prefix*, then the `s3` protocol selects all files that start with the specified prefix as data files for the external table. The `s3` protocol does not use the slash character (`/`) as a delimiter, so a slash character following a prefix is treated as part of the prefix itself.

For example, consider the following 5 files that each have the *S3_endpoint* named `s3-us-west-2.amazonaws.com` and the *bucket_name* `test1`:

```
s3://s3-us-west-2.amazonaws.com/test1/abc
s3://s3-us-west-2.amazonaws.com/test1/abc/
s3://s3-us-west-2.amazonaws.com/test1/abc/xx
s3://s3-us-west-2.amazonaws.com/test1/abcdef
s3://s3-us-west-2.amazonaws.com/test1/abcdefff
```

- If the S3 URL is provided as `s3://s3-us-west-2.amazonaws.com/test1/abc`, then the `abc` prefix selects all 5 files.
- If the S3 URL is provided as `s3://s3-us-west-2.amazonaws.com/test1/abc/`, then the `abc/` prefix selects the files `s3://s3-us-west-2.amazonaws.com/test1/abc/` and `s3://s3-us-west-2.amazonaws.com/test1/abc/xx`.
- If the S3 URL is provided as `s3://s3-us-west-2.amazonaws.com/test1/abcd`, then the `abcd` prefix selects the files `s3://s3-us-west-2.amazonaws.com/test1/abcdef` and `s3://s3-us-west-2.amazonaws.com/test1/abcdefff`

Wildcard characters are not supported in an *S3_prefix*; however, the S3 prefix functions as if a wildcard character immediately followed the prefix itself.

All of the files selected by the S3 URL (*S3_endpoint*/*bucket_name*/*S3_prefix*) are used as the source for the external table, so they must have the same format. Each file must also contain complete data rows. A data row cannot be split between files. The S3 file permissions must be `Open/Download` and `View` for the S3 user ID that is accessing the files.

For information about the Amazon S3 endpoints see *http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region*. For information about S3 buckets and folders, see the Amazon S3 documentation *https://aws.amazon.com/documentation/s3/*. For information about the S3 file prefix, see the Amazon S3 documentation *Listing Keys Hierarchically Using a Prefix and Delimiter*.

The `config` parameter specifies the location of the required `s3` protocol configuration file that contains AWS connection credentials and communication parameters. See *About the s3 Protocol config Parameter*.

About S3 Data Files

For each `INSERT` operation to a writable S3 table, each Greenplum Database segment uploads a single file to the configured S3 bucket using the filename format `<prefix><segment_id><random>.<extension>[.gz]` where:

- `<prefix>` is the prefix specified in the S3 URL.
- `<segment_id>` is the Greenplum Database segment ID.
- `<random>` is a random number that is used to ensure that the filename is unique.
- `<extension>` describes the file type (`.txt` or `.csv`, depending on the value you provide in the `FORMAT` clause of `CREATE WRITABLE EXTERNAL TABLE`). Files created by the `gpcheckcloud` utility always uses the extension `.data`.

- `.gz` is appended to the filename if compression is enabled for S3 writable tables (the default).

For writable S3 tables, you can configure the buffer size and the number of threads that segments use for uploading files. See *s3 Protocol Configuration File*.

For read-only S3 tables, all of the files specified by the S3 file location (*S3_endpoint*/*bucket_name*/*S3_prefix*) are used as the source for the external table and must have the same format. Each file must also contain complete data rows. If the files contain an optional header row, the column names in the header row cannot contain a newline character (`\n`) or a carriage return (`\r`). Also, the column delimiter cannot be a newline character (`\n`) or a carriage return character (`\r`).

For read-only S3 tables, the `s3` protocol recognizes gzip and deflate compressed files and automatically decompresses the files. For gzip compression, the protocol recognizes the format of a gzip compressed file. For deflate compression, the protocol assumes a file with the `.deflate` suffix is a deflate compressed file.

The S3 file permissions must be `Open/Download` and `View` for the S3 user ID that is accessing the files. Writable S3 tables require the S3 user ID to have `Upload/Delete` permissions.

For read-only S3 tables, each segment can download one file at a time from S3 location using several threads. To take advantage of the parallel processing performed by the Greenplum Database segments, the files in the S3 location should be similar in size and the number of files should allow for multiple segments to download the data from the S3 location. For example, if the Greenplum Database system consists of 16 segments and there was sufficient network bandwidth, creating 16 files in the S3 location allows each segment to download a file from the S3 location. In contrast, if the location contained only 1 or 2 files, only 1 or 2 segments download data.

s3 Protocol AWS Server-Side Encryption Support

Greenplum Database supports server-side encryption using Amazon S3-managed keys (SSE-S3) for AWS S3 files you access with readable and writable external tables created using the `s3` protocol. SSE-S3 encrypts your object data as it writes to disk, and transparently decrypts the data for you when you access it.

> **Note:** The `s3` protocol supports SSE-S3 only for Amazon Web Services S3 files. SS3-SE is not supported when accessing files in S3 compatible services.

Your S3 `accessid` and `secret` permissions govern your access to all S3 bucket objects, whether the data is encrypted or not. However, you must configure your client to use S3-managed keys for accessing encrypted data.

Refer to *Protecting Data Using Server-Side Encryption* in the AWS documentation for additional information about AWS Server-Side Encryption.

Configuring S3 Server-Side Encryption

`s3` protocol server-side encryption is disabled by default. To take advantage of server-side encryption on AWS S3 objects you write using the Greenplum Database `s3` protocol, you must set the `server_side_encryption` configuration parameter in your `s3` configuration file to the value `sse-s3`:

```
server_side_encryption = sse-s3
```

When the configuration file you provide to a `CREATE WRITABLE EXTERNAL TABLE` call using the `s3` protocol includes the `server_side_encryption = sse-s3` setting, Greenplum Database applies encryption headers for you on all `INSERT` operations on that external table. S3 then encrypts on write the object(s) identified by the URI you provided in the `LOCATION` clause.

S3 transparently decrypts data during read operations of encrypted files accessed via readable external tables you create using the `s3` protocol. No additional configuration is required.

For further encryption configuration granularity, you may consider creating Amazon Web Services S3 *Bucket Policy*(s), identifying the objects you want to encrypt and the write actions on those objects as

described in the *Protecting Data Using Server-Side Encryption with Amazon S3-Managed Encryption Keys (SSE-S3)* AWS documentation.

s3 Protocol Proxy Support

You can specify a URL that is the proxy that S3 uses to connect to a data source. S3 supports these protocols: HTTP and HTTPS. You can specify a proxy with the `s3` protocol configuration parameter `proxy` or an environment variable. If the configuration parameter is set, the environment variables are ignored.

To specify proxy with an environment variable, you set the environment variable based on the protocol: `http_proxy` or `https_proxy`. You can specify a different URL for each protocol by setting the appropriate environment variable. S3 supports these environment variables.

- `all_proxy` specifies the proxy URL that is used if an environment variable for a specific protocol is not set.
- `no_proxy` specifies a comma-separated list of hosts names that do not use the proxy specified by an environment variable.

The environment variables must be set must and must be accessible to Greenplum Database on all Greenplum Database hosts.

For information about the configuration parameter `proxy`, see *s3 Protocol Configuration File*.

About the s3 Protocol config Parameter

The optional `config` parameter specifies the location of the required `s3` protocol configuration file. The file contains Amazon Web Services (AWS) connection credentials and communication parameters. For information about the file, see *s3 Protocol Configuration File*.

The configuration file is required on all Greenplum Database segment hosts. This is default location is a location in the data directory of each Greenplum Database segment instance.

```
gpseg_data_dir/gpseg_prefixN/s3/s3.conf
```

The *gpseg_data_dir* is the path to the Greenplum Database segment data directory, the *gpseg_prefix* is the segment prefix, and *N* is the segment ID. The segment data directory, prefix, and ID are set when you initialize a Greenplum Database system.

If you have multiple segment instances on segment hosts, you can simplify the configuration by creating a single location on each segment host. Then you specify the absolute path to the location with the `config` parameter in the `s3` protocol `LOCATION` clause. This example specifies a location in the `gpadmin` home directory.

```
LOCATION ('s3://s3-us-west-2.amazonaws.com/test/my_data config=/home/
gpadmin/s3.conf')
```

All segment instances on the hosts use the file `/home/gpadmin/s3.conf`.

s3 Protocol Configuration File

When using the `s3` protocol, an `s3` protocol configuration file is required on all Greenplum Database segments. The default location is:

```
gpseg_data_dir/gpseg-prefixN/s3/s3.conf
```

The *gpseg_data_dir* is the path to the Greenplum Database segment data directory, the *gpseg-prefix* is the segment prefix, and *N* is the segment ID. The segment data directory, prefix, and ID are set when you initialize a Greenplum Database system.

If you have multiple segment instances on segment hosts, you can simplify the configuration by creating a single location on each segment host. Then you can specify the absolute path to the location with the

`config` parameter in the `s3` protocol `LOCATION` clause. However, note that both read-only and writable S3 external tables use the same parameter values for their connections. If you want to configure protocol parameters differently for read-only and writable S3 tables, then you must use two different `s3` protocol configuration files and specify the correct file in the `CREATE EXTERNAL TABLE` statement when you create each table.

This example specifies a single file location in the `s3` directory of the `gpadmin` home directory:

```
config=/home/gpadmin/s3/s3.conf
```

All segment instances on the hosts use the file `/home/gpadmin/s3/s3.conf`.

The `s3` protocol configuration file is a text file that consists of a `[default]` section and parameters This is an example configuration file:

```
[default]
secret = "secret"
accessid = "user access id"
threadnum = 3
chunksize = 67108864
```

You can use the Greenplum Database `gpcheckcloud` utility to test the S3 configuration file. See *Using the gpcheckcloud Utility*.

s3 Configuration File Parameters

`accessid`

Required. AWS S3 ID to access the S3 bucket.

`secret`

Required. AWS S3 passcode for the S3 ID to access the S3 bucket.

`autocompress`

For writable S3 external tables, this parameter specifies whether to compress files (using gzip) before uploading to S3. Files are compressed by default if you do not specify this parameter.

`chunksize`

The buffer size that each segment thread uses for reading from or writing to the S3 server. The default is 64 MB. The minimum is 8MB and the maximum is 128MB.

When inserting data to a writable S3 table, each Greenplum Database segment writes the data into its buffer (using multiple threads up to the `threadnum` value) until it is full, after which it writes the buffer to a file in the S3 bucket. This process is then repeated as necessary on each segment until the insert operation completes.

Because Amazon S3 allows a maximum of 10,000 parts for multipart uploads, the minimum `chunksize` value of 8MB supports a maximum insert size of 80GB per Greenplum database segment. The maximum `chunksize` value of 128MB supports a maximum insert size 1.28TB per segment. For writable S3 tables, you must ensure that the `chunksize` setting can support the anticipated table size of your table. See *Multipart Upload Overview* in the S3 documentation for more information about uploads to S3.

`encryption`

Use connections that are secured with Secure Sockets Layer (SSL). Default value is `true`. The values `true`, `t`, `on`, `yes`, and `y` (case insensitive) are treated as `true`. Any other value is treated as `false`.

If the port is not specified in the URL in the `LOCATION` clause of the `CREATE EXTERNAL TABLE` command, the configuration file `encryption` parameter affects the port used by

the `s3` protocol (port 80 for HTTP or port 443 for HTTPS). If the port is specified, that port is used regardless of the encryption setting.

`gpcheckcloud_newline`

When downloading files from an S3 location, the `gpcheckcloud` utility appends a new line character to last line of a file if the last line of a file does not have an EOL (end of line) character. The default character is `\n` (newline). The value can be `\n`, `\r` (carriage return), or `\n\r` (newline/carriage return).

Adding an EOL character prevents the last line of one file from being concatenated with the first line of next file.

`low_speed_limit`

The upload/download speed lower limit, in bytes per second. The default speed is 10240 (10K). If the upload or download speed is slower than the limit for longer than the time specified by `low_speed_time`, then the connection is aborted and retried. After 3 retries, the `s3` protocol returns an error. A value of 0 specifies no lower limit.

`low_speed_time`

When the connection speed is less than `low_speed_limit`, this parameter specified the amount of time, in seconds, to wait before aborting an upload to or a download from the S3 bucket. The default is 60 seconds. A value of 0 specifies no time limit.

`proxy`

Specify a URL that is the proxy that S3 uses to connect to a data source. S3 supports these protocols: HTTP and HTTPS. This is the format for the parameter.

```
proxy = protocol://[user:password@]proxyhost[:port]
```

If this parameter is not set or is an empty string (`proxy = ""`), S3 uses the proxy specified by the environment variable `http_proxy` or `https_proxy` (and the environment variables `all_proxy` and `no_proxy`). The environment variable that S3 uses depends on the protocol. For information about the environment variables, see *s3 Protocol Proxy Support* in the *Greenplum Database Administrator Guide*.

There can be at most one `proxy` parameter in the configuration file. The URL specified by the parameter is the proxy for all supported protocols.

`server_side_encryption`

The S3 server-side encryption method that has been configured for the bucket. Greenplum Database supports only server-side encryption with Amazon S3-managed keys, identified by the configuration parameter value `sse-s3`. Server-side encryption is disabled (`none`) by default.

`threadnum`

The maximum number of concurrent threads a segment can create when uploading data to or downloading data from the S3 bucket. The default is 4. The minimum is 1 and the maximum is 8.

`verifycert`

Controls how the `s3` protocol handles authentication when establishing encrypted communication between a client and an S3 data source over HTTPS. The value is either `true` or `false`. The default value is `true`.

- `verifycert=false` - Ignores authentication errors and allows encrypted communication over HTTPS.
- `verifycert=true` - Requires valid authentication (a proper certificate) for encrypted communication over HTTPS.

Setting the value to `false` can be useful in testing and development environments to allow communication without changing certificates.

> **Warning:** Setting the value to `false` exposes a security risk by ignoring invalid credentials when establishing communication between a client and a S3 data store.

`version`

Specifies the version of the information specified in the `LOCATION` clause of the `CREATE EXTERNAL TABLE` command. The value is either `1` or `2`. The default value is `1`.

If the value is `1`, the `LOCATION` clause supports an Amazon S3 URL, and does not contain the `region` parameter. If the value is `2`, the `LOCATION` clause supports S3 compatible services and must include the `region` parameter. The `region` parameter specifies the S3 data source region. For this S3 URL `s3://s3-us-west-2.amazonaws.com/s3test.example.com/dataset1/normal/`, the AWS S3 region is `us-west-2`.

If `version` is 1 or is not specified, this is an example of the `LOCATION` clause of the `CREATE EXTERNAL TABLE` command that specifies an Amazon S3 endpoint.

```
LOCATION ('s3://s3-us-west-2.amazonaws.com/s3test.example.com/
dataset1/normal/ config=/home/gpadmin/aws_s3/s3.conf')
```

If `version` is 2, this is an example `LOCATION` clause with the `region` parameter for an AWS S3 compatible service.

```
LOCATION ('s3://test.company.com/s3test.company/test1/normal/
 region=local-test config=/home/gpadmin/aws_s3/s3.conf')
```

If `version` is 2, the `LOCATION` clause can also specify an Amazon S3 endpoint. This example specifies an Amazon S3 endpoint that uses the `region` parameter.

```
LOCATION ('s3://s3-us-west-2.amazonaws.com/s3test.example.com/
dataset1/normal/ region=us-west-2 config=/home/gpadmin/aws_s3/
s3.conf')
```

Note: Greenplum Database can require up to `threadnum * chunksize` memory on each segment host when uploading or downloading S3 files. Consider this `s3` protocol memory requirement when you configure overall Greenplum Database memory.

s3 Protocol Limitations

These are `s3` protocol limitations:

- Only the S3 path-style URL is supported.

  ```
  s3://S3_endpoint/bucketname/[S3_prefix]
  ```

- Only the S3 endpoint is supported. The protocol does not support virtual hosting of S3 buckets (binding a domain name to an S3 bucket).
- AWS signature version 4 signing process is supported.

  For information about the S3 endpoints supported by each signing process, see *http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region*.
- Only a single URL and optional configuration file is supported in the `LOCATION` clause of the `CREATE EXTERNAL TABLE` command.
- If the `NEWLINE` parameter is not specified in the `CREATE EXTERNAL TABLE` command, the newline character must be identical in all data files for specific prefix. If the newline character is different in some data files with the same prefix, read operations on the files might fail.

- For writable S3 external tables, only the `INSERT` operation is supported. `UPDATE`, `DELETE`, and `TRUNCATE` operations are not supported.
- Because Amazon S3 allows a maximum of 10,000 parts for multipart uploads, the maximum `chunksize` value of 128MB supports a maximum insert size of 1.28TB per Greenplum database segment for writable s3 tables. You must ensure that the `chunksize` setting can support the anticipated table size of your table. See *Multipart Upload Overview* in the S3 documentation for more information about uploads to S3.
- To take advantage of the parallel processing performed by the Greenplum Database segment instances, the files in the S3 location for read-only S3 tables should be similar in size and the number of files should allow for multiple segments to download the data from the S3 location. For example, if the Greenplum Database system consists of 16 segments and there was sufficient network bandwidth, creating 16 files in the S3 location allows each segment to download a file from the S3 location. In contrast, if the location contained only 1 or 2 files, only 1 or 2 segments download data.

Using the gpcheckcloud Utility

The Greenplum Database utility `gpcheckcloud` helps users create an `s3` protocol configuration file and test a configuration file. You can specify options to test the ability to access an S3 bucket with a configuration file, and optionally upload data to or download data from files in the bucket.

If you run the utility without any options, it sends a template configuration file to `STDOUT`. You can capture the output and create an `s3` configuration file to connect to Amazon S3.

The utility is installed in the Greenplum Database `$GPHOME/bin` directory.

Syntax

```
gpcheckcloud {-c | -d} "s3://S3_endpoint/bucketname/[S3_prefix]
 [config=path_to_config_file]"

gpcheckcloud -u <file_to_upload> "s3://S3_endpoint/bucketname/[S3_prefix]
 [config=path_to_config_file]"
gpcheckcloud -t

gpcheckcloud -h
```

Options

`-c`

Connect to the specified S3 location with the configuration specified in the `s3` protocol URL and return information about the files in the S3 location.

If the connection fails, the utility displays information about failures such as invalid credentials, prefix, or server address (DNS error), or server not available.

`-d`

Download data from the specified S3 location with the configuration specified in the `s3` protocol URL and send the output to `STDOUT`.

If files are gzip compressed or have a `.deflate` suffix to indicate deflate compression, the uncompressed data is sent to `STDOUT`.

`-u`

Upload a file to the S3 bucket specified in the `s3` protocol URL using the specified configuration file if available. Use this option to test compression and `chunksize` and `autocompress` settings for your configuration.

`-t`

Sends a template configuration file to `STDOUT`. You can capture the output and create an `s3` configuration file to connect to Amazon S3.

`-h`

Display `gpcheckcloud` help.

Examples

This example runs the utility without options to create a template `s3` configuration file `mytest_s3.config` in the current directory.

```
gpcheckcloud -t > ./mytest_s3.config
```

This example attempts to upload a local file, `test-data.csv` to an S3 bucket location using the `s3` configuration file `s3.mytestconf`:

```
gpcheckcloud -u ./test-data.csv "s3://s3-us-west-2.amazonaws.com/test1/abc
 config=s3.mytestconf"
```

A successful upload results in one or more files placed in the S3 bucket using the filename format `abc<segment_id><random>.data[.gz]`. See *About S3 Data Files*.

This example attempts to connect to an S3 bucket location with the `s3` configuration file `s3.mytestconf`.

```
gpcheckcloud -c "s3://s3-us-west-2.amazonaws.com/test1/abc
 config=s3.mytestconf"
```

Download all files from the S3 bucket location and send the output to `STDOUT`.

```
gpcheckcloud -d "s3://s3-us-west-2.amazonaws.com/test1/abc
 config=s3.mytestconf"
```

Using a Custom Protocol

A custom protocol allows you to connect Greenplum Database to a data source that cannot be accessed with the `file://`, `gpfdist://`, or `pxf://` protocols.

Creating a custom protocol requires that you implement a set of C functions with specified interfaces, declare the functions in Greenplum Database, and then use the `CREATE TRUSTED PROTOCOL` command to enable the protocol in the database.

See *Example Custom Data Access Protocol* for an example.

Handling Errors in External Table Data

By default, if external table data contains an error, the command fails and no data loads into the target database table.

Define the external table with single row error handling to enable loading correctly formatted rows and to isolate data errors in external table data. See *Handling Load Errors*.

The `gpfdist` file server uses the `HTTP` protocol. External table queries that use `LIMIT` end the connection after retrieving the rows, causing an HTTP socket error. If you use `LIMIT` in queries of external tables that use the `gpfdist://` or `http:// protocols`, ignore these errors – data is returned to the database as expected.

Creating and Using External Web Tables

External web tables allow Greenplum Database to treat dynamic data sources like regular database tables. Because web table data can change as a query runs, the data is not rescannable.

`CREATE EXTERNAL WEB TABLE` creates a web table definition. You can define command-based or URL-based external web tables. The definition forms are distinct: you cannot mix command-based and URL-based definitions.

Command-based External Web Tables

The output of a shell command or script defines command-based web table data. Specify the command in the EXECUTE clause of CREATE EXTERNAL WEB TABLE. The data is current as of the time the command runs. The EXECUTE clause runs the shell command or script on the specified master, and/or segment host or hosts. The command or script must reside on the hosts corresponding to the host(s) defined in the EXECUTE clause.

By default, the command is run on segment hosts when active segments have output rows to process. For example, if each segment host runs four primary segment instances that have output rows to process, the command runs four times per segment host. You can optionally limit the number of segment instances that execute the web table command. All segments included in the web table definition in the ON clause run the command in parallel.

The command that you specify in the external table definition executes from the database and cannot access environment variables from .bashrc or .profile. Set environment variables in the EXECUTE clause. For example:

```
=# CREATE EXTERNAL WEB TABLE output (output text)
    EXECUTE 'PATH=/home/gpadmin/programs; export PATH; myprogram.sh'
    FORMAT 'TEXT';
```

Scripts must be executable by the gpadmin user and reside in the same location on the master or segment hosts.

The following command defines a web table that runs a script. The script runs on each segment host where a segment has output rows to process.

```
=# CREATE EXTERNAL WEB TABLE log_output
    (linenum int, message text)
    EXECUTE '/var/load_scripts/get_log_data.sh' ON HOST
    FORMAT 'TEXT' (DELIMITER '|');
```

URL-based External Web Tables

A URL-based web table accesses data from a web server using the HTTP protocol. Web table data is dynamic; the data is not rescannable.

Specify the LOCATION of files on a web server using http://. The web data file(s) must reside on a web server that Greenplum segment hosts can access. The number of URLs specified corresponds to the number of segment instances that work in parallel to access the web table. For example, if you specify two external files to a Greenplum Database system with eight primary segments, two of the eight segments access the web table in parallel at query runtime.

The following sample command defines a web table that gets data from several URLs.

```
=# CREATE EXTERNAL WEB TABLE ext_expenses (name text,
  date date, amount float4, category text, description text)
  LOCATION (

  'http://intranet.company.com/expenses/sales/file.csv',
  'http://intranet.company.com/expenses/exec/file.csv',
  'http://intranet.company.com/expenses/finance/file.csv',
  'http://intranet.company.com/expenses/ops/file.csv',
  'http://intranet.company.com/expenses/marketing/file.csv',
  'http://intranet.company.com/expenses/eng/file.csv'

   )
  FORMAT 'CSV' ( HEADER );
```

Examples for Creating External Tables

These examples show how to define external data with different protocols. Each `CREATE EXTERNAL TABLE` command can contain only one protocol.

> **Note:** When using IPv6, always enclose the numeric IP addresses in square brackets.

Start `gpfdist` before you create external tables with the `gpfdist` protocol. The following code starts the `gpfdist` file server program in the background on port *8081* serving files from directory `/var/data/staging`. The logs are saved in `/home/gpadmin/log`.

```
gpfdist -p 8081 -d /var/data/staging -l /home/gpadmin/log &
```

Example 1—Single gpfdist instance on single-NIC machine

Creates a readable external table, `ext_expenses`, using the `gpfdist` protocol. The files are formatted with a pipe (|) as the column delimiter.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
    date date, amount float4, category text, desc1 text )
    LOCATION ('gpfdist://etlhost-1:8081/*')
FORMAT 'TEXT' (DELIMITER '|');
```

Example 2—Multiple gpfdist instances

Creates a readable external table, *ext_expenses,* using the `gpfdist` protocol from all files with the *txt* extension. The column delimiter is a pipe (|) and NULL (' ') is a space.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*.txt',
             'gpfdist://etlhost-2:8081/*.txt')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ') ;
```

Example 3—Multiple gpfdists instances

Creates a readable external table, *ext_expenses,* from all files with the *txt* extension using the `gpfdists` protocol. The column delimiter is a pipe (|) and NULL (' ') is a space. For information about the location of security certificates, see *gpfdists:// Protocol*.

1. Run `gpfdist` with the `--ssl` option.
2. Run the following command.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gpfdists://etlhost-1:8081/*.txt',
             'gpfdists://etlhost-2:8082/*.txt')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ') ;
```

Example 4—Single gpfdist instance with error logging

Uses the gpfdist protocol to create a readable external table, `ext_expenses,` from all files with the *txt* extension. The column delimiter is a pipe (|) and NULL (' ') is a space.

Access to the external table is single row error isolation mode. Input data formatting errors are captured internally in Greenplum Database with a description of the error. See *Viewing Bad Rows in the Error Log* for information about investigating error rows. You can view the errors, fix the issues, and then reload the

rejected data. If the error count on a segment is greater than five (the `SEGMENT REJECT LIMIT` value), the entire external table operation fails and no rows are processed.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date, amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*.txt',
             'gpfdist://etlhost-2:8082/*.txt')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
   LOG ERRORS SEGMENT REJECT LIMIT 5;
```

To create the readable ext_expenses table from CSV-formatted text files:

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*.txt',
             'gpfdist://etlhost-2:8082/*.txt')
   FORMAT 'CSV' ( DELIMITER ',' )
   LOG ERRORS SEGMENT REJECT LIMIT 5;
```

Example 5—TEXT Format on a Hadoop Distributed File Server

Creates a readable external table, *ext_expenses,* using the `pxf` protocol. The column delimiter is a pipe (`|`).

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('pxf://dir/data/filename.txt?PROFILE=hdfs:text')
   FORMAT 'TEXT' (DELIMITER '|');
```

Refer to *Accessing External Data with PXF* for information about using the Greenplum Platform Extension Framework (PXF) to access data on a Hadoop Distributed File System.

Example 6—Multiple files in CSV format with header rows

Creates a readable external table, *ext_expenses,* using the `file` protocol. The files are `CSV` format and have a header row.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('file://filehost/data/international/*',
             'file://filehost/data/regional/*',
             'file://filehost/data/supplement/*.csv')
   FORMAT 'CSV' (HEADER);
```

Example 7—Readable External Web Table with Script

Creates a readable external web table that executes a script once per segment host:

```
=# CREATE EXTERNAL WEB TABLE log_output (linenum int,
    message text)
   EXECUTE '/var/load_scripts/get_log_data.sh' ON HOST
   FORMAT 'TEXT' (DELIMITER '|');
```

Example 8—Writable External Table with gpfdist

Creates a writable external table, *sales_out,* that uses `gpfdist` to write output data to the file *sales.out*. The column delimiter is a pipe (|) and NULL (' ') is a space. The file will be created in the directory specified when you started the gpfdist file server.

```
=# CREATE WRITABLE EXTERNAL TABLE sales_out (LIKE sales)
   LOCATION ('gpfdist://etl1:8081/sales.out')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
   DISTRIBUTED BY (txn_id);
```

Example 9—Writable External Web Table with Script

Creates a writable external web table, `campaign_out`, that pipes output data received by the segments to an executable script, `to_adreport_etl.sh`:

```
=# CREATE WRITABLE EXTERNAL WEB TABLE campaign_out
    (LIKE campaign)
    EXECUTE '/var/unload_scripts/to_adreport_etl.sh'
    FORMAT 'TEXT' (DELIMITER '|');
```

Example 10—Readable and Writable External Tables with XML Transformations

Greenplum Database can read and write XML data to and from external tables with gpfdist. For information about setting up an XML transform, see *Transforming External Data with gpfdist and gpload*.

Accessing External Data with Foreign Tables

Greenplum Database implements portions of the SQL/MED specification, allowing you to access data that resides outside of Greenplum using regular SQL queries. Such data is referred to as *foreign* or external data.

You can access foreign data with help from a *foreign-data wrapper*. A foreign-data wrapper is a library that communicates with a remote data source. This library hides the source-specific connection and data access details.

> **Note:** Most PostgreSQL foreign-data wrappers should work with Greenplum Database. However, PostgreSQL foreign-data wrappers connect only through the Greenplum Database master and do not access the Greenplum Database segment instances directly.

If none of the existing foreign-data wrappers suit your needs, you can write your own as described in *Writing a Foreign Data Wrapper*.

To access foreign data, you create a *foreign server* object, which defines how to connect to a particular remote data source according to the set of options used by its supporting foreign-data wrapper. Then you create one or more *foreign tables*, which define the structure of the remote data. A foreign table can be used in queries just like a normal table, but a foreign table has no storage in the Greenplum Database server. Whenever a foreign table is accessed, Greenplum Database asks the foreign-data wrapper to fetch data from, or update data in (if supported by the wrapper), the remote source.

> **Note:** The Pivotal Query Optimizer, GPORCA, does not support foreign tables. A query on a foreign table always falls back to the Postgres Planner.

Accessing remote data may require authenticating to the remote data source. This information can be provided by a *user mapping*, which can provide additional data such as a user name and password based on the current Greenplum Database role.

For additional information, refer to *`CREATE FOREIGN DATA WRAPPER`*, *`CREATE SERVER`*, *`CREATE USER MAPPING`*, and *`CREATE FOREIGN TABLE`*.

Writing a Foreign Data Wrapper

This chapter outlines how to write a new foreign-data wrapper.

All operations on a foreign table are handled through its foreign-data wrapper (FDW), a library that consists of a set of functions that the core Greenplum Database server calls. The foreign-data wrapper is responsible for fetching data from the remote data store and returning it to the Greenplum Database executor. If updating foreign-data is supported, the wrapper must handle that, too.

The foreign-data wrappers included in the Greenplum Database open source github repository are good references when trying to write your own. You may want to examine the `file_fdw` and `postgres_fdw` modules in the `contrib/` directory. The `CREATE FOREIGN DATA WRAPPER` reference page also provides some useful details.

> **Note:** The SQL standard specifies an interface for writing foreign-data wrappers. Greenplum Database does not implement that API, however, because the effort to accommodate it into Greenplum would be large, and the standard API hasn't yet gained wide adoption.

This topic includes the following sections:

Requirements

When you develop with the Greenplum Database foreign-data wrapper API:

- You must develop your code on a system with the same hardware and software architecture as that of your Greenplum Database hosts.
- Your code must be written in a compiled language such as C, using the version-1 interface. For details on C language calling conventions and dynamic loading, refer to *C Language Functions* in the PostgreSQL documentation.
- Symbol names in your object files must not conflict with each other nor with symbols defined in the Greenplum Database server. You must rename your functions or variables if you get error messages to this effect.
- Review the foreign table introduction described in *Accessing External Data with Foreign Tables*.

Known Issues and Limitations

The Greenplum Database 6 foreign-data wrapper implementation has the following known issues and limitations:

- The Greenplum Database 6 distribution does not install any foreign data wrappers.
- Greenplum Database uses the `mpp_execute` option value for foreign table scans only. Greenplum does not honor the `mpp_execute` setting when you write to, or update, a foreign table; all write operations are initiated through the master.

Header Files

The Greenplum Database header files that you may use when you develop a foreign-data wrapper are located in the `greenplum-db/src/include/` directory (when developing against the Greenplum

Database open source github repository), or installed in the `$GPHOME/include/postgresql/server/` directory (when developing against a Greenplum installation):

- foreign/fdwapi.h - FDW API structures and callback function signatures
- foreign/foreign.h - foreign-data wrapper helper structs and functions
- catalog/pg_foreign_table.h - foreign table definition
- catalog/pg_foreign_server.h - foreign server definition

Your FDW code may also be dependent on header files and libraries required to access the remote data store.

Foreign Data Wrapper Functions

The developer of a foreign-data wrapper must implement an SQL-invokable *handler* function, and optionally an SQL-invokable *validator* function. Both functions must be written in a compiled language such as C, using the version-1 interface.

The *handler* function simply returns a struct of function pointers to callback functions that will be called by the Greenplum Database planner, executor, and various maintenance commands. The *handler* function must be registered with Greenplum Database as taking no arguments and returning the special pseudo-type `fdw_handler`. For example:

```
CREATE FUNCTION NEW_fdw_handler()
  RETURNS fdw_handler
  AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Most of the effort in writing a foreign-data wrapper is in implementing the callback functions. The FDW API callback functions, plain C functions that are not visible or callable at the SQL level, are described in *Foreign Data Wrapper Callback Functions*.

The *validator* function is responsible for validating options provided in `CREATE` and `ALTER` commands for its foreign-data wrapper, as well as foreign servers, user mappings, and foreign tables using the wrapper. The *validator* function must be registered as taking two arguments, a text array containing the options to be validated, and an OID representing the type of object with which the options are associated. For example:

```
CREATE FUNCTION NEW_fdw_validator( text[], oid )
  RETURNS void
  AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

The OID argument reflects the type of the system catalog that the object would be stored in, one of `ForeignDataWrapperRelationId`, `ForeignServerRelationId`, `UserMappingRelationId`, or `ForeignTableRelationId`. If no *validator* function is supplied by a foreign data wrapper, Greenplum Database does not check option validity at object creation time or object alteration time.

Foreign Data Wrapper Callback Functions

The foreign-data wrapper API defines callback functions that Greenplum Database invokes when scanning and updating foreign tables. The API also includes callbacks for performing explain and analyze operations on a foreign table.

The *handler* function of a foreign-data wrapper returns a `palloc`'d `FdwRoutine` struct containing pointers to callback functions described below. The `FdwRoutine` struct is located in the `foreign/fdwapi.h` header file, and is defined as follows:

```
/*
 * FdwRoutine is the struct returned by a foreign-data wrapper's handler
 * function.  It provides pointers to the callback functions needed by the
 * planner and executor.
```

```
 *
 * More function pointers are likely to be added in the future.  Therefore
 * it's recommended that the handler initialize the struct with
 * makeNode(FdwRoutine) so that all fields are set to NULL.  This will
 * ensure that no fields are accidentally left undefined.
 */
typedef struct FdwRoutine
{
 NodeTag  type;

 /* Functions for scanning foreign tables */
 GetForeignRelSize_function GetForeignRelSize;
 GetForeignPaths_function GetForeignPaths;
 GetForeignPlan_function GetForeignPlan;
 BeginForeignScan_function BeginForeignScan;
 IterateForeignScan_function IterateForeignScan;
 ReScanForeignScan_function ReScanForeignScan;
 EndForeignScan_function EndForeignScan;

 /*
  * Remaining functions are optional.  Set the pointer to NULL for any that
  * are not provided.
  */

 /* Functions for updating foreign tables */
 AddForeignUpdateTargets_function AddForeignUpdateTargets;
 PlanForeignModify_function PlanForeignModify;
 BeginForeignModify_function BeginForeignModify;
 ExecForeignInsert_function ExecForeignInsert;
 ExecForeignUpdate_function ExecForeignUpdate;
 ExecForeignDelete_function ExecForeignDelete;
 EndForeignModify_function EndForeignModify;
 IsForeignRelUpdatable_function IsForeignRelUpdatable;

 /* Support functions for EXPLAIN */
 ExplainForeignScan_function ExplainForeignScan;
 ExplainForeignModify_function ExplainForeignModify;

 /* Support functions for ANALYZE */
 AnalyzeForeignTable_function AnalyzeForeignTable;
} FdwRoutine;
```

You must implement the scan-related functions in your foreign-data wrapper; implementing the other callback functions is optional.

Scan-related callback functions include:

Callback Signature	Description
`void GetForeignRelSize (PlannerInfo *root, RelOptInfo *baserel, Oid foreigntableid)`	Obtain relation size estimates for a foreign table. Called at the beginning of planning for a query on a foreign table.
`void GetForeignPaths (PlannerInfo *root, RelOptInfo *baserel, Oid foreigntableid)`	Create possible access paths for a scan on a foreign table. Called during query planning. **Note:** A Greenplum Database-compatible FDW must call `create_foreignscan_path()` in its `GetForeignPaths()` callback function.
`ForeignScan * GetForeignPlan (PlannerInfo *root, RelOptInfo *baserel, Oid foreigntableid, ForeignPath *best_path, List *tlist, List *scan_clauses)`	Create a `ForeignScan` plan node from the selected foreign access path. Called at the end of query planning.
`void BeginForeignScan (ForeignScanState *node, int eflags)`	Begin executing a foreign scan. Called during executor startup.
`TupleTableSlot * IterateForeignScan (ForeignScanState *node)`	Fetch one row from the foreign source, returning it in a tuple table slot; return NULL if no more rows are available.
`void ReScanForeignScan (ForeignScanState *node)`	Restart the scan from the beginning.
`void`	End the scan and release resources.

Callback Signature	Description
`EndForeignScan (ForeignScanState *node)`	

Refer to *Foreign Data Wrapper Callback Routines* in the PostgreSQL documentation for detailed information about the inputs and outputs of the FDW callback functions.

Foreign Data Wrapper Helper Functions

The FDW API exports several helper functions from the Greenplum Database core server so that authors of foreign-data wrappers have easy access to attributes of FDW-related objects, such as options provided when the user creates or alters the foreign-data wrapper, server, or foreign table. To use these helper functions, you must include `foreign.h` header file in your source file:

```
#include "foreign/foreign.h"
```

The FDW API includes the helper functions listed in the table below. Refer to *Foreign Data Wrapper Helper Functions* in the PostgreSQL documentation for more information about these functions.

Helper Signature	Description
`ForeignDataWrapper * GetForeignDataWrapper(Oid fdwid);`	Returns the `ForeignDataWrapper` object for the foreign-data wrapper with the given OID.
`ForeignDataWrapper * GetForeignDataWrapperByName(const char *name, bool missing_ok);`	Returns the `ForeignDataWrapper` object for the foreign-data wrapper with the given name.
`ForeignServer * GetForeignServer(Oid serverid);`	Returns the `ForeignServer` object for the foreign server with the given OID.
`ForeignServer * GetForeignServerByName(const char *name, bool missing_ok);`	Returns the `ForeignServer` object for the foreign server with the given name.
`UserMapping * GetUserMapping(Oid userid, Oid serverid);`	Returns the `UserMapping` object for the user mapping of the given role on the given server.
`ForeignTable * GetForeignTable(Oid relid);`	Returns the `ForeignTable` object for the foreign table with the given OID.
`List * GetForeignColumnOptions(Oid relid, AttrNumber attnum);`	Returns the per-column FDW options for the column with the given foreign table OID and attribute number.

Greenplum Database Considerations

A Greenplum Database user can specify the `mpp_execute` option when they create or alter a foreign table, foreign server, or foreign data wrapper. A Greenplum Database-compatible foreign-data wrapper examines the `mpp_execute` option value on a scan and uses it to determine where to request data - from the `master` (the default), `any` (master or any one segment), or `all` segments.

> **Note:** Write/update operations using a foreign data wrapper are always executed on the Greenplum Database master, regardless of the `mpp_execute` setting.

The following scan code snippet probes the `mpp_execute` value associated with a foreign table:

```
ForeignTable *table = GetForeignTable(foreigntableid);
if (table->exec_location == FTEXECLOCATION_ALL_SEGMENTS)
{
    ...
}
```

```
else if (table->exec_location == FTEXECLOCATION_ANY)
{
    ...
}
else if (table->exec_location == FTEXECLOCATION_MASTER)
{
    ...
}
```

If the foreign table was not created with an `mpp_execute` option setting, the `mpp_execute` setting of the foreign server, and then the foreign data wrapper, is probed and used. If none of the foreign-data-related objects has an `mpp_execute` setting, the default setting is `master`.

If a foreign-data wrapper supports `mpp_execute 'all'`, it will implement a policy that matches Greenplum segments to data. So as not to duplicate data retrieved from the remote, the FDW on each segment must be able to establish which portion of the data is their responsibility. An FDW may use the segment identifier and the number of segments to help make this determination. The following code snippet demonstrates how a foreign-data wrapper may retrieve the segment number and total number of segments:

```
int segmentNumber = GpIdentity.segindex;
int totalNumberOfSegments = getgpsegmentCount();
```

Building a Foreign Data Wrapper Extension with PGXS

You compile the foreign-data wrapper functions that you write with the FDW API into one or more shared libraries that the Greenplum Database server loads on demand.

You can use the PostgreSQL build extension infrastructure (PGXS) to build the source code for your foreign-data wrapper against a Greenplum Database installation. This framework automates common build rules for simple modules. If you have a more complicated use case, you will need to write your own build system.

To use the PGXS infrastructure to generate a shared library for your FDW, create a simple `Makefile` that sets PGXS-specific variables.

> **Note:** Refer to *Extension Building Infrastructure* in the PostgreSQL documentation for information about the `Makefile` variables supported by PGXS.

For example, the following `Makefile` generates a shared library in the current working directory named `base_fdw.so` from two C source files, base_fdw_1.c and base_fdw_2.c:

```
MODULE_big = base_fdw
OBJS = base_fdw_1.o base_fdw_2.o

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)

PG_CPPFLAGS = -I$(shell $(PG_CONFIG) --includedir)
SHLIB_LINK = -L$(shell $(PG_CONFIG) --libdir)
include $(PGXS)
```

A description of the directives used in this `Makefile` follows:

- `MODULE_big` - identifes the base name of the shared library generated by the `Makefile`
- `PG_CPPFLAGS` - adds the Greenplum Database installation `include/` directory to the compiler header file search path
- `SHLIB_LINK` adds the Greenplum Database installation library directory (`$GPHOME/lib/`) to the linker search path
- The `PG_CONFIG` and `PGXS` variable settings and the `include` statement are required and typically reside in the last three lines of the `Makefile`.

To package the foreign-data wrapper as a Greenplum Database extension, you create script (*newfdw--version*.sql) and control (*newfdw*.control) files that register the FDW *handler* and *validator* functions, create the foreign data wrapper, and identify the characteristics of the FDW shared library file.

> **Note:** *Packaging Related Objects into an Extension* in the PostgreSQL documentation describes how to package an extension.

Example foreign-data wrapper extension script file named base_fdw--1.0.sql:

```
CREATE FUNCTION base_fdw_handler()
  RETURNS fdw_handler
  AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

CREATE FUNCTION base_fdw_validator(text[], oid)
  RETURNS void
  AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

CREATE FOREIGN DATA WRAPPER base_fdw
  HANDLER base_fdw_handler
  VALIDATOR base_fdw_validator;
```

Example FDW control file named base_fdw.control:

```
# base_fdw FDW extension
comment = 'base foreign-data wrapper implementation; does not do much'
default_version = '1.0'
module_pathname = '$libdir/base_fdw'
relocatable = true
```

When you add the following directives to the Makefile, you identify the FDW extension control file base name (EXTENSION) and SQL script (DATA):

```
EXTENSION = base_fdw
DATA = base_fdw--1.0.sql
```

Running make install with these directives in the Makefile copies the shared library and FDW SQL and control files into the specified or default locations in your Greenplum Database installation ($GPHOME).

Deployment Considerations

You must package the FDW shared library and extension files in a form suitable for deployment in a Greenplum Database cluster. When you construct and deploy the package, take into consideration the following:

- The FDW shared library must be installed to the same file system location on the master host and on every segment host in the Greenplum Database cluster. You specify this location in the .control file. This location is typically the $GPHOME/lib/postgresql/ directory.
- The FDW .sql and .control files must be installed to the $GPHOME/share/postgresql/extension/ directory on the master host and on every segment host in the Greenplum Database cluster.
- The gpadmin user must have permission to traverse the complete file system path to the FDW shared library file and extension files.

Using the Greenplum Parallel File Server (gpfdist)

The `gpfdist` protocol is used in a `CREATE EXTERNAL TABLE` SQL command to access external data served by the Greenplum Database `gpfdist` file server utility. When external data is served by `gpfdist`, all segments in the Greenplum Database system can read or write external table data in parallel.

This topic describes the setup and management tasks for using `gpfdist` with external tables.

About gpfdist and External Tables

The `gpfdist` file server utility is located in the `$GPHOME/bin` directory on your Greenplum Database master host and on each segment host. When you start a `gpfdist` instance you specify a listen port and the path to a directory containing files to read or where files are to be written. For example, this command runs `gpfdist` in the background, listening on port 8801, and serving files in the `/home/gpadmin/external_files` directory:

```
$ gpfdist -p 8801 -d /home/gpadmin/external_files &
```

The `CREATE EXTERNAL TABLE` command `LOCATION` clause connects an external table definition to one or more `gpfdist` instances. If the external table is readable, the `gpfdist` server reads data records from files from in specified directory, packs them into a block, and sends the block in a response to a Greenplum Database segment's request. The segments unpack rows they receive and distribute them according to the external table's distribution policy. If the external table is a writable table, segments send blocks of rows in a request to gpfdist and gpfdist writes them to the external file.

External data files can contain rows in CSV format or any delimited text format supported by the `FORMAT` clause of the `CREATE EXTERNAL TABLE` command. In addition, `gpfdist` can be configured with a YAML-formatted file to transform external data files between a supported text format and another format, for example XML or JSON. See <ref> for an example that shows how to use `gpfdist` to read external XML files into a Greenplum Database readable external table.

For readable external tables, `gpfdist` uncompresses `gzip` (`.gz`) and `bzip2` (`.bz2`) files automatically. You can use the wildcard character (*) or other C-style pattern matching to denote multiple files to read. External files are assumed to be relative to the directory specified when you started the `gpfdist` instance.

About gpfdist Setup and Performance

You can run `gpfdist` instances on multiple hosts and you can run multiple `gpfdist` instances on each host. This allows you to deploy `gpfdist` servers strategically so that you can attain fast data load and unload rates by utilizing all of the available network bandwidth and Greenplum Database's parallelism.

- Allow network traffic to use all ETL host network interfaces simultaneously. Run one instance of `gpfdist` for each interface on the ETL host, then declare the host name of each NIC in the `LOCATION` clause of your external table definition (see *Examples for Creating External Tables*).

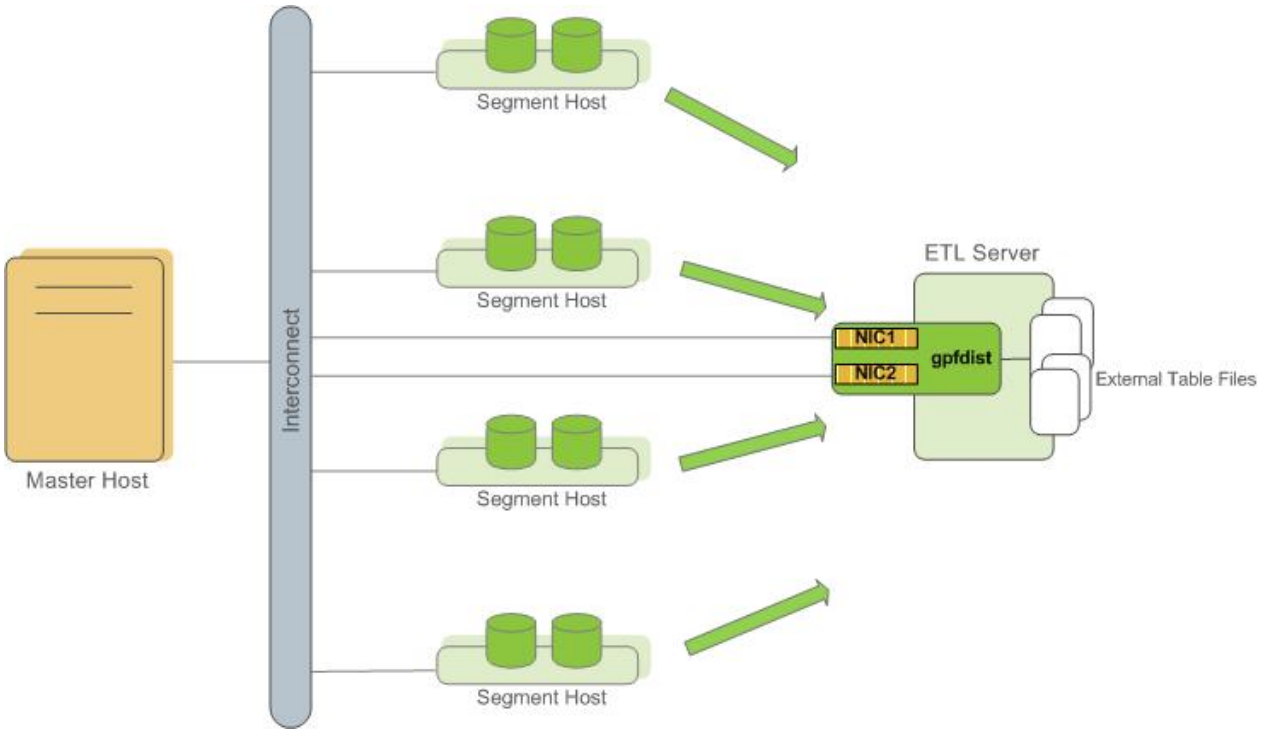


Figure 25: External Table Using Single gpfdist Instance with Multiple NICs

- Divide external table data equally among multiple `gpfdist` instances on the ETL host. For example, on an ETL system with two NICs, run two `gpfdist` instances (one on each NIC) to optimize data load performance and divide the external table data files evenly between the two `gpfdist` servers.

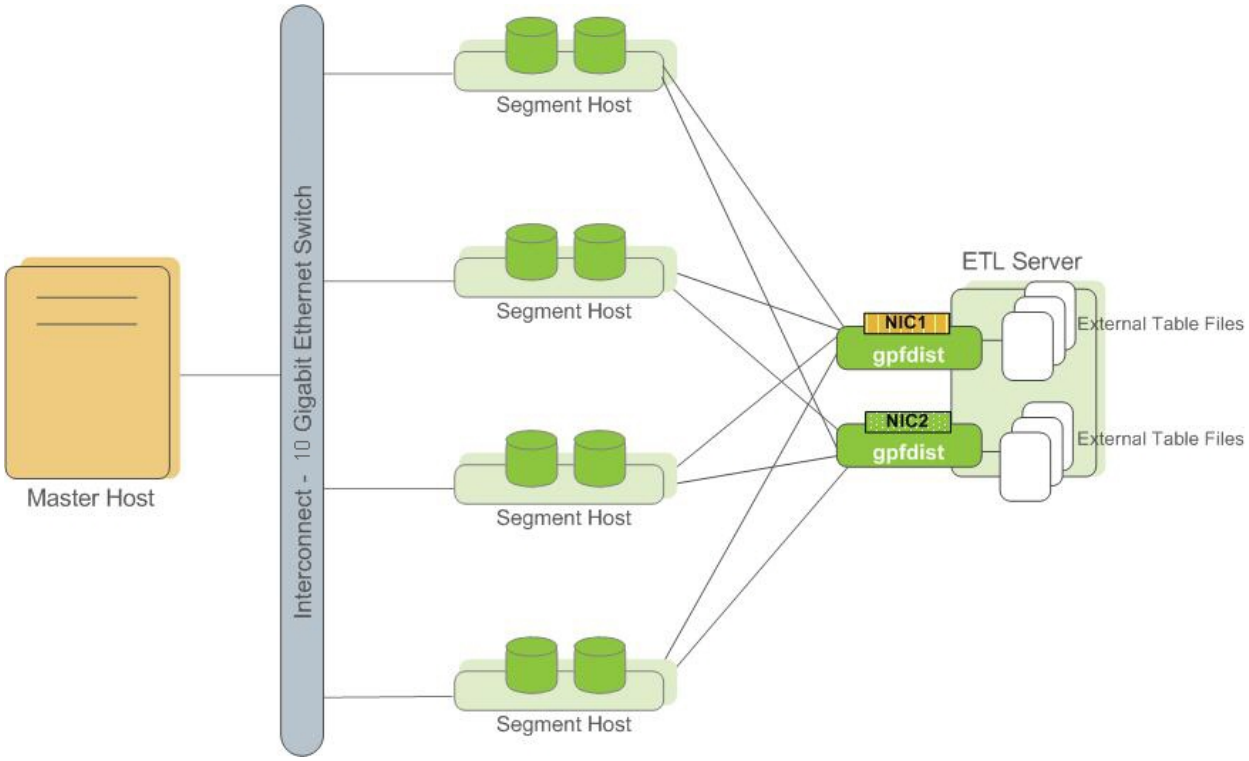


Figure 26: External Tables Using Multiple gpfdist Instances with Multiple NICs

Note: Use pipes (|) to separate formatted text when you submit files to `gpfdist`. Greenplum Database encloses comma-separated text strings in single or double quotes. `gpfdist` has to remove the quotes to parse the strings. Using pipes to separate formatted text avoids the extra step and improves performance.

Controlling Segment Parallelism

The `gp_external_max_segs` server configuration parameter controls the number of segment instances that can access a single `gpfdist` instance simultaneously. 64 is the default. You can set the number of segments such that some segments process external data files and some perform other database processing. Set this parameter in the `postgresql.conf` file of your master instance.

Installing gpfdist

`gpfdist` is installed in `$GPHOME/bin` of your Greenplum Database master host installation. Run `gpfdist` on a machine other than the Greenplum Database master or standby master, such as on a machine devoted to ETL processing. Running `gpfdist` on the master or standby master can have a performance impact on query execution. To install `gpfdist` on your ETL server, get it from the *Greenplum Load Tools* package and follow its installation instructions.

Starting and Stopping gpfdist

You can start `gpfdist` in your current directory location or in any directory that you specify. The default port is `8080`.

From your current directory, type:

```
gpfdist &
```

From a different directory, specify the directory from which to serve files, and optionally, the HTTP port to run on.

To start `gpfdist` in the background and log output messages and errors to a log file:

```
$ gpfdist -d /var/load_files -p 8081 -l /home/gpadmin/log &
```

For multiple `gpfdist` instances on the same ETL host (see *Figure 25: External Table Using Single gpfdist Instance with Multiple NICs*), use a different base directory and port for each instance. For example:

```
$ gpfdist -d /var/load_files1 -p 8081 -l /home/gpadmin/log1 &
$ gpfdist -d /var/load_files2 -p 8082 -l /home/gpadmin/log2 &
```

To stop `gpfdist` when it is running in the background:

First find its process id:

```
$ ps -ef | grep gpfdist
```

Then kill the process, for example (where 3456 is the process ID in this example):

```
$ kill 3456
```

Troubleshooting gpfdist

The segments access `gpfdist` at runtime. Ensure that the Greenplum segment hosts have network access to `gpfdist`. `gpfdist` is a web server: test connectivity by running the following command from each host in the Greenplum array (segments and master):

```
$ wget http://gpfdist_hostname:port/filename
```

The `CREATE EXTERNAL TABLE` definition must have the correct host name, port, and file names for `gpfdist`. Specify file names and paths relative to the directory from which `gpfdist` serves files (the directory path specified when `gpfdist` started). See *Examples for Creating External Tables*.

If you start `gpfdist` on your system and IPv6 networking is disabled, `gpfdist` displays this warning message when testing for an IPv6 port.

```
[WRN gpfdist.c:2050] Creating the socket failed
```

If the corresponding IPv4 port is available, `gpfdist` uses that port and the warning for IPv6 port can be ignored. To see information about the ports that `gpfdist` tests, use the `-V` option.

For information about IPv6 and IPv4 networking, see your operating system documentation.

When reading or writing data with the `gpfdist` or `gfdists` protocol, the `gpfdist` utility rejects HTTP requests that do not include `X-GP-PROTO` in the request header. If `X-GP-PROTO` is not detected in the header request `gpfist` returns a 400 error in the status line of the HTTP response header: `400 invalid request (no gp-proto)`.

Greenplum Database includes `X-GP-PROTO` in the HTTP request header to indicate that the request is from Greenplum Database.

If the `gpfdist` utility hangs with no read or write activity occurring, you can generate a core dump the next time a hang occurs to help debug the issue. Set the environment variable `GPFDIST_WATCHDOG_TIMER` to the number of seconds of no activity to wait before `gpfdist` is forced to exit. When the environment variable is set and `gpfdist` hangs, the utility aborts after the specified number of seconds, creates a core dump, and sends abort information to the log file.

This example sets the environment variable on a Linux system so that `gpfdist` exits after 300 seconds (5 minutes) of no activity.

```
export GPFDIST_WATCHDOG_TIMER=300
```

Loading and Unloading Data

The topics in this section describe methods for loading and writing data into and out of a Greenplum Database, and how to format data files.

Greenplum Database supports high-performance parallel data loading and unloading, and for smaller amounts of data, single file, non-parallel data import and export.

Greenplum Database can read from and write to several types of external data sources, including text files, Hadoop file systems, Amazon S3, and web servers.

- The `COPY` SQL command transfers data between an external text file on the master host, or multiple text files on segment hosts, and a Greenplum Database table.
- Readable external tables allow you to query data outside of the database directly and in parallel using SQL commands such as `SELECT`, `JOIN`, or `SORT EXTERNAL TABLE DATA`, and you can create views for external tables. External tables are often used to load external data into a regular database table using a command such as `CREATE TABLE` *`table`* `AS SELECT * FROM` *`ext_table`*.
- External web tables provide access to dynamic data. They can be backed with data from URLs accessed using the HTTP protocol or by the output of an OS script running on one or more segments.
- The `gpfdist` utility is the Greenplum Database parallel file distribution program. It is an HTTP server that is used with external tables to allow Greenplum Database segments to load external data in parallel, from multiple file systems. You can run multiple instances of `gpfdist` on different hosts and network interfaces and access them in parallel.
- The `gpload` utility automates the steps of a load task using `gpfdist` and a YAML-formatted control file.
- You can create readable and writable external tables with the Greenplum Platform Extension Framework (PXF), and use these tables to load data into, or offload data from, Greenplum Database. For information about using PXF, refer to *Accessing External Data with PXF*.
- The Greenplum-Kafka Integration provides high-speed, parallel data transfer from Kafka to Greenplum Database. For information about using these tools, refer to the *Greenplum-Kafka Integration* documentation.
- The Greenplum Streaming Server is an ETL tool and API that you can use to load data into Greenplum Database. For information about using this tool, refer to the *Greenplum Streaming Server* documentation.
- The Greenplum-Spark Connector provides high speed, parallel data transfer between Pivotal Greenplum Database and Apache Spark. For information about using the Greenplum-Spark Connector, refer to the documentation at *https://greenplum-spark.docs.pivotal.io/*.
- The Greenplum-Informatica Connector provides high speed data transfer from an Informatica PowerCenter cluster to a Pivotal Greenplum Database cluster for batch and streaming ETL operations. For information about using the Greenplum-Informatica Connector, refer to the documentation at *https://greenplum-informatica.docs.pivotal.io/*.

The method you choose to load data depends on the characteristics of the source data—its location, size, format, and any transformations required.

In the simplest case, the `COPY` SQL command loads data into a table from a text file that is accessible to the Greenplum Database master instance. This requires no setup and provides good performance for smaller amounts of data. With the `COPY` command, the data copied into or out of the database passes between a single file on the master host and the database. This limits the total size of the dataset to the capacity of the file system where the external file resides and limits the data transfer to a single file write stream.

More efficient data loading options for large datasets take advantage of the Greenplum Database MPP architecture, using the Greenplum Database segments to load data in parallel. These methods allow data to load simultaneously from multiple file systems, through multiple NICs, on multiple hosts, achieving very high data transfer rates. External tables allow you to access external files from within the database

as if they are regular database tables. When used with `gpfdist`, the Greenplum Database parallel file distribution program, external tables provide full parallelism by using the resources of all Greenplum Database segments to load or unload data.

Greenplum Database leverages the parallel architecture of the Hadoop Distributed File System to access files on that system.

Loading Data Using an External Table

Use SQL commands such as `INSERT` and `SELECT` to query a readable external table, the same way that you query a regular database table. For example, to load travel expense data from an external table, `ext_expenses`, into a database table, `expenses_travel`:

```
=# INSERT INTO expenses_travel
    SELECT * from ext_expenses where category='travel';
```

To load all data into a new database table:

```
=# CREATE TABLE expenses AS SELECT * from ext_expenses;
```

Loading and Writing Non-HDFS Custom Data

Greenplum Database supports `TEXT` and `CSV` formats for importing and exporting data through external tables. You can load and save data in other formats by defining a custom format or custom protocol or by setting up a transformation with the `gpfdist` parallel file server.

Using a Custom Format

You specify a custom data format in the `FORMAT` clause of `CREATE EXTERNAL TABLE`.

```
FORMAT 'CUSTOM' (formatter=format_function, key1=val1,...keyn=valn)
```

Where the `'CUSTOM'` keyword indicates that the data has a custom format and `formatter` specifies the function to use to format the data, followed by comma-separated parameters to the formatter function.

Greenplum Database provides functions for formatting fixed-width data, but you must author the formatter functions for variable-width data. The steps are as follows.

1. Author and compile input and output functions as a shared library.
2. Specify the shared library function with `CREATE FUNCTION` in Greenplum Database.
3. Use the `formatter` parameter of `CREATE EXTERNAL TABLE`'s `FORMAT` clause to call the function.

Importing and Exporting Fixed Width Data

Specify custom formats for fixed-width data with the Greenplum Database functions `fixedwith_in` and `fixedwidth_out`. These functions already exist in the file `$GPHOME/share/postgresql/cdb_external_extensions.sql`. The following example declares a custom format, then calls the `fixedwidth_in` function to format the data.

```
CREATE READABLE EXTERNAL TABLE students (
name varchar(20), address varchar(30), age int)
LOCATION ('file://<host>/file/path/')
FORMAT 'CUSTOM' (formatter=fixedwidth_in,
         name='20', address='30', age='4');
```

The following options specify how to import fixed width data.

- Read all the data.

To load all the fields on a line of fixed with data, you must load them in their physical order. You must specify the field length, but cannot specify a starting and ending position. The fields names in the fixed width arguments must match the order in the field list at the beginning of the `CREATE TABLE` command.

- Set options for blank and null characters.

  Trailing blanks are trimmed by default. To keep trailing blanks, use the `preserve_blanks=on` option. You can reset the trailing blanks option to the default with the `preserve_blanks=off` option.

  Use the `null='null_string_value'` option to specify a value for null characters.
- If you specify `preserve_blanks=on`, you must also define a value for null characters.
- If you specify `preserve_blanks=off`, null is not defined, and the field contains only blanks, Greenplum writes a null to the table. If null is defined, Greenplum writes an empty string to the table.

  Use the `line_delim='line_ending'` parameter to specify the line ending character. The following examples cover most cases. The `E` specifies an escape string constant.

```
line_delim=E'\n'
line_delim=E'\r'
line_delim=E'\r\n'
line_delim='abc'
```

Examples: Read Fixed-Width Data

The following examples show how to read fixed-width data.

Example 1 – Loading a table with all fields defined

```
CREATE READABLE EXTERNAL TABLE students (
name varchar(20), address varchar(30), age int)
LOCATION ('file://<host>/file/path/')
FORMAT 'CUSTOM' (formatter=fixedwidth_in,
         name=20, address=30, age=4);
```

Example 2 – Loading a table with PRESERVED_BLANKS on

```
CREATE READABLE EXTERNAL TABLE students (
name varchar(20), address varchar(30), age int)
LOCATION ('gpfdist://<host>:<portNum>/file/path/')
FORMAT 'CUSTOM' (formatter=fixedwidth_in,
         name=20, address=30, age=4,
        preserve_blanks='on',null='NULL');
```

Example 3 – Loading data with no line delimiter

```
CREATE READABLE EXTERNAL TABLE students (
name varchar(20), address varchar(30), age int)
LOCATION ('file://<host>/file/path/')
FORMAT 'CUSTOM' (formatter=fixedwidth_in,
         name='20', address='30', age='4', line_delim='?@')
```

Example 4 – Create a writable external table with a \r\n line delimiter

```
CREATE WRITABLE EXTERNAL TABLE students_out (
name varchar(20), address varchar(30), age int)
LOCATION ('gpfdist://<host>:<portNum>/file/path/students_out.txt')
FORMAT 'CUSTOM' (formatter=fixedwidth_out,
```

```
        name=20, address=30, age=4, line_delim=E'\r\n');
```

Using a Custom Protocol

Greenplum Database provides protocols such as `gpfdist`, `http`, and `file` for accessing data over a network, or you can author a custom protocol. You can use the standard data formats, `TEXT` and `CSV`, or a custom data format with custom protocols.

You can create a custom protocol whenever the available built-in protocols do not suffice for a particular need. For example, you could connect Greenplum Database in parallel to another system directly, and stream data from one to the other without the need to materialize the data on disk or use an intermediate process such as `gpfdist`. You must be a superuser to create and register a custom protocol.

1. Author the send, receive, and (optionally) validator functions in C, with a predefined API. These functions are compiled and registered with the Greenplum Database. For an example custom protocol, see *Example Custom Data Access Protocol* .
2. After writing and compiling the read and write functions into a shared object (.so), declare a database function that points to the .so file and function names.

   The following examples use the compiled import and export code.

   ```
   CREATE FUNCTION myread() RETURNS integer
   as '$libdir/gpextprotocol.so', 'myprot_import'
   LANGUAGE C STABLE;
   CREATE FUNCTION mywrite() RETURNS integer
   as '$libdir/gpextprotocol.so', 'myprot_export'
   LANGUAGE C STABLE;
   ```

   The format of the optional validator function is:

   ```
   CREATE OR REPLACE FUNCTION myvalidate() RETURNS void
   AS '$libdir/gpextprotocol.so', 'myprot_validate'
   LANGUAGE C STABLE;
   ```
3. Create a protocol that accesses these functions. `Validatorfunc` is optional.

   ```
   CREATE TRUSTED PROTOCOL myprot(
   writefunc='mywrite',
   readfunc='myread',
   validatorfunc='myvalidate');
   ```
4. Grant access to any other users, as necessary.

   ```
   GRANT ALL ON PROTOCOL myprot TO otheruser;
   ```
5. Use the protocol in readable or writable external tables.

   ```
   CREATE WRITABLE EXTERNAL TABLE ext_sales(LIKE sales)
   LOCATION ('myprot://<meta>/<meta>/…')
   FORMAT 'TEXT';
   CREATE READABLE EXTERNAL TABLE ext_sales(LIKE sales)
   LOCATION('myprot://<meta>/<meta>/…')
   FORMAT 'TEXT';
   ```

Declare custom protocols with the SQL command `CREATE TRUSTED PROTOCOL`, then use the `GRANT` command to grant access to your users. For example:

- Allow a user to create a readable external table with a trusted protocol

  ```
  GRANT SELECT ON PROTOCOL <protocol name> TO <user name>;
  ```

- Allow a user to create a writable external table with a trusted protocol

```
GRANT INSERT ON PROTOCOL <protocol name> TO <user name>;
```

- Allow a user to create readable and writable external tables with a trusted protocol

```
GRANT ALL ON PROTOCOL <protocol name> TO <user name>;
```

Handling Load Errors

Readable external tables are most commonly used to select data to load into regular database tables. You use the `CREATE TABLE AS SELECT` or `INSERT INTO` commands to query the external table data. By default, if the data contains an error, the entire command fails and the data is not loaded into the target database table.

The `SEGMENT REJECT LIMIT` clause allows you to isolate format errors in external table data and to continue loading correctly formatted rows. Use `SEGMENT REJECT LIMIT` to set an error threshold, specifying the reject limit `count` as number of `ROWS` (the default) or as a `PERCENT` of total rows (1-100).

The entire external table operation is aborted, and no rows are processed, if the number of error rows reaches the `SEGMENT REJECT LIMIT`. The limit of error rows is per-segment, not per entire operation. The operation processes all good rows, and it discards and optionally logs formatting errors for erroneous rows, if the number of error rows does not reach the `SEGMENT REJECT LIMIT`.

The `LOG ERRORS` clause allows you to keep error rows for further examination. For information about the `LOG ERRORS` clause, see the `CREATE EXTERNAL TABLE` command in the *Greenplum Database Reference Guide*.

When you set `SEGMENT REJECT LIMIT`, Greenplum scans the external data in single row error isolation mode. Single row error isolation mode applies to external data rows with format errors such as extra or missing attributes, attributes of a wrong data type, or invalid client encoding sequences. Greenplum does not check constraint errors, but you can filter constraint errors by limiting the `SELECT` from an external table at runtime. For example, to eliminate duplicate key errors:

```
=# INSERT INTO table_with_pkeys
    SELECT DISTINCT * FROM external_table;
```

Note: When loading data with the `COPY` command or an external table, the value of the server configuration parameter `gp_initial_bad_row_limit` limits the initial number of rows that are processed that are not formatted properly. The default is to stop processing if the first 1000 rows contain formatting errors. See the *Greenplum Database Reference Guide* for information about the parameter.

Define an External Table with Single Row Error Isolation

The following example logs errors internally in Greenplum Database and sets an error threshold of 10 errors.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*',
             'gpfdist://etlhost-2:8082/*')
   FORMAT 'TEXT' (DELIMITER '|')
   LOG ERRORS SEGMENT REJECT LIMIT 10
     ROWS;
```

Use the built-in SQL function `gp_read_error_log('external_table')` to read the error log data. This example command displays the log errors for *ext_expenses*:

```
SELECT gp_read_error_log('ext_expenses');
```

For information about the format of the error log, see *Viewing Bad Rows in the Error Log*.

The built-in SQL function `gp_truncate_error_log('external_table')` deletes the error data. This example deletes the error log data created from the previous external table example :

```
SELECT gp_truncate_error_log('ext_expenses');
```

Capture Row Formatting Errors and Declare a Reject Limit

The following SQL fragment captures formatting errors internally in Greenplum Database and declares a reject limit of 10 rows.

```
LOG ERRORS SEGMENT REJECT LIMIT 10 ROWS
```

Use the built-in SQL function `gp_read_error_log()` to read the error log data. For information about viewing log errors, see *Viewing Bad Rows in the Error Log*.

Viewing Bad Rows in the Error Log

If you use single row error isolation (see *Define an External Table with Single Row Error Isolation* or *Running COPY in Single Row Error Isolation Mode*), any rows with formatting errors are logged internally by Greenplum Database.

Greenplum Database captures the following error information in a table format:

Table 57: Error Log Format

column	type	description
cmdtime	timestamptz	Timestamp when the error occurred.
relname	text	The name of the external table or the target table of a COPY command.
filename	text	The name of the load file that contains the error.
linenum	int	If COPY was used, the line number in the load file where the error occurred. For external tables using file:// protocol or gpfdist:// protocol and CSV format, the file name and line number is logged.
bytenum	int	For external tables with the gpfdist:// protocol and data in TEXT format: the byte offset in the load file where the error occurred. gpfdist parses TEXT files in blocks, so logging a line number is not possible. CSV files are parsed a line at a time so line number tracking is possible for CSV files.
errmsg	text	The error message text.
rawdata	text	The raw data of the rejected row.

column	type	description
rawbytes	bytea	In cases where there is a database encoding error (the client encoding used cannot be converted to a server-side encoding), it is not possible to log the encoding error as *rawdata*. Instead the raw bytes are stored and you will see the octal code for any non seven bit ASCII characters.

You can use the Greenplum Database built-in SQL function `gp_read_error_log()` to display formatting errors that are logged internally. For example, this command displays the error log information for the table *ext_expenses*:

```
SELECT gp_read_error_log('ext_expenses');
```

For information about managing formatting errors that are logged internally, see the command `COPY` or `CREATE EXTERNAL TABLE` in the *Greenplum Database Reference Guide*.

Moving Data between Tables

You can use `CREATE TABLE AS` or `INSERT...SELECT` to load external and external web table data into another (non-external) database table, and the data will be loaded in parallel according to the external or external web table definition.

If an external table file or external web table data source has an error, one of the following will happen, depending on the isolation mode used:

- **Tables without error isolation mode**: any operation that reads from that table fails. Loading from external and external web tables without error isolation mode is an all or nothing operation.
- **Tables with error isolation mode**: the entire file will be loaded, except for the problematic rows (subject to the configured REJECT_LIMIT)

Loading Data with gpload

The Greenplum `gpload` utility loads data using readable external tables and the Greenplum parallel file server (`gpfdist` or `gpfdists`). It handles parallel file-based external table setup and allows users to configure their data format, external table definition, and `gpfdist` or `gpfdists` setup in a single configuration file.

> **Note:** `gpfdist` and `gpload` are compatible only with the Greenplum Database major version in which they are shipped. For example, a `gpfdist` utility that is installed with Greenplum Database 4.x cannot be used with Greenplum Database 5.x or 6.x.
>
> **Note:** `MERGE` and `UPDATE` operations are not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

To use gpload

1. Ensure that your environment is set up to run `gpload`. Some dependent files from your Greenplum Database installation are required, such as `gpfdist` and Python, as well as network access to the Greenplum segment hosts.

   See the *Greenplum Database Reference Guide* for details.
2. Create your load control file. This is a YAML-formatted file that specifies the Greenplum Database connection information, `gpfdist` configuration information, external table options, and data format.

   See the *Greenplum Database Reference Guide* for details.

For example:

```
---
VERSION: 1.0.0.1
DATABASE: ops
USER: gpadmin
HOST: mdw-1
PORT: 5432
GPLOAD:
   INPUT:
    - SOURCE:
         LOCAL_HOSTNAME:
           - etl1-1
           - etl1-2
           - etl1-3
           - etl1-4
         PORT: 8081
         FILE:
           - /var/load/data/*
    - COLUMNS:
           - name: text
           - amount: float4
           - category: text
           - descr: text
           - date: date
    - FORMAT: text
    - DELIMITER: '|'
    - ERROR_LIMIT: 25
    - LOG_ERRORS: true
   OUTPUT:
    - TABLE: payables.expenses
    - MODE: INSERT
   PRELOAD:
    - REUSE_TABLES: true
SQL:
   - BEFORE: "INSERT INTO audit VALUES('start', current_timestamp)"
   - AFTER: "INSERT INTO audit VALUES('end', current_timestamp)"
```

3. Run `gpload`, passing in the load control file. For example:

```
gpload -f my_load.yml
```

Accessing External Data with PXF

Data managed by your organization may already reside in external sources such as Hadoop, object stores, and other SQL databases. The Greenplum Platform Extension Framework (PXF) provides access to this external data via built-in connectors that map an external data source to a Greenplum Database table definition.

PXF is installed with Hadoop and Object Storage connectors. These connectors enable you to read external data stored in text, Avro, JSON, RCFile, Parquet, SequenceFile, and ORC formats. You can use the JDBC connector to access an external SQL database.

> **Note:** In previous versions of Greenplum Database, you may have used the `gphdfs` external table protocol to access data stored in Hadoop. Greenplum Database version 6.0.0 removes the `gphdfs` protocol. Use PXF and the `pxf` external table protocol to access Hadoop in Greenplum Database version 6.x.

The Greenplum Platform Extension Framework includes a C-language extension and a Java service. After you configure and initialize PXF, you start a single PXF JVM process on each Greenplum Database segment host. This long- running process concurrently serves multiple query requests.

For detailed information about the architecture of and using PXF, refer to the *Greenplum Platform Extension Framework (PXF)* documentation.

Transforming External Data with gpfdist and gpload

The `gpfdist` parallel file server allows you to set up transformations that enable Greenplum Database external tables to read and write files in formats that are not supported with the `CREATE EXTERNAL TABLE` command's `FORMAT` clause. An *input* transformation reads a file in the foreign data format and outputs rows to `gpfdist` in the CSV or other text format specified in the external table's `FORMAT` clause. An *output* transformation receives rows from `gpfdist` in text format and converts them to the foreign data format.

> **Note:** `gpfdist` and `gpload` are compatible only with the Greenplum Database major version in which they are shipped. For example, a `gpfdist` utility that is installed with Greenplum Database 4.x cannot be used with Greenplum Database 5.x or 6.x.

This topic describes the tasks to set up data transformations that work with `gpfdist` to read or write external data files with formats that Greenplum Database does not support.

- *About gpfdist Transformations*
- *Determine the Transformation Schema*
- *Write a Transformation*
- *Write the gpfdist Configuration File*
- *Transfer the Data*
- *Configuration File Format*
- *XML Transformation Examples*

About gpfdist Transformations

To set up a transformation for a data format, you provide an executable command that `gpfdist` can call with the name of the file containing data. For example, you could write a shell script that runs an XSLT transformation on an XML file to output rows with columns delimited with a vertical bar (`|`) character and rows delimited with linefeeds.

Transformations are configured in a YAML-formatted configuration file passed to `gpfdist` on the command line.

If you want to load the external data into a table in the Greenplum database, you can use the `gpload` utility to automate the tasks to create an external table, run `gpfdist`, and load the transformed data into the database table.

Accessing data in external XML files from within the database is a common example requiring transformation. The following diagram shows *gpfdist* performing a transformation on XML files on an ETL server.

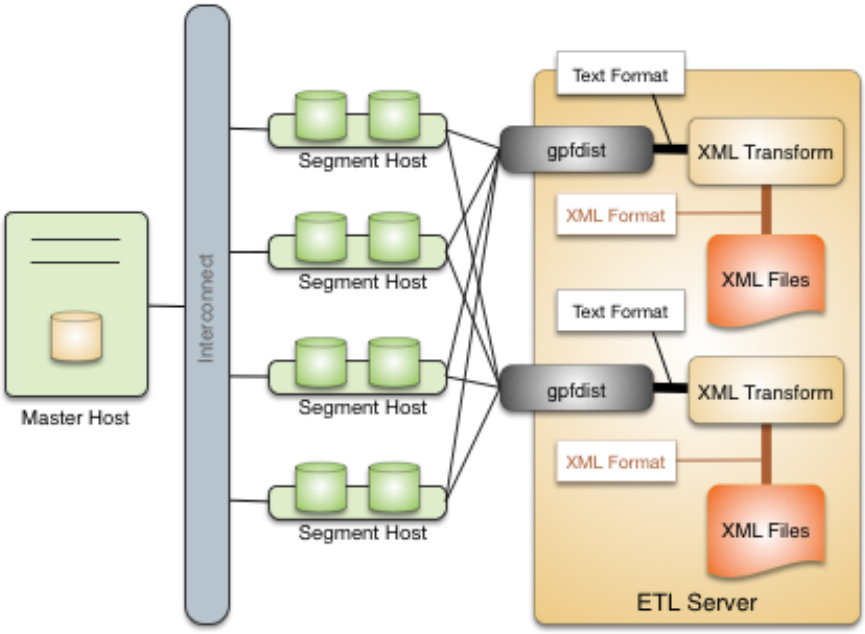


Figure 27: External Tables using XML Transformations

Following are the high-level steps to set up a `gpfdist` transformation for external data files. The process is illustrated with an XML example.

1. *Determine the transformation schema*.
2. *Write a transformation*.
3. *Write the gpfdist configuration file*.
4. *Transfer the data*.

Determine the Transformation Schema

To prepare for the transformation project:

1. Determine the goal of the project, such as indexing data, analyzing data, combining data, and so on.
2. Examine the source files and note the file structure and element names.
3. Choose the elements to import and decide if any other limits are appropriate.

For example, the following XML file, *prices.xml*, is a simple XML file that contains price records. Each price record contains two fields: an item number and a price.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<prices>
  <pricerecord>
    <itemnumber>708421</itemnumber>
    <price>19.99</price>
  </pricerecord>
  <pricerecord>
    <itemnumber>708466</itemnumber>
    <price>59.25</price>
  </pricerecord>
  <pricerecord>
    <itemnumber>711121</itemnumber>
    <price>24.99</price>
  </pricerecord>
</prices>
```

The goal of this transformation is to import all the data into a Greenplum Database readable external table with an integer `itemnumber` column and a decimal `price` column.

Write a Transformation

The transformation specifies what to extract from the data. You can use any authoring environment and language appropriate for your project. For XML transformations choose from technologies such as XSLT, Joost (STX), Java, Python, or Perl, based on the goals and scope of the project.

In the price example, the next step is to transform the XML data into a two-column delimited text format.

```
708421|19.99
708466|59.25
711121|24.99
```

The following STX transform, called *input_transform.stx*, performs the data transformation.

```
<?xml version="1.0"?>
<stx:transform version="1.0"
   xmlns:stx="http://stx.sourceforge.net/2002/ns"
   pass-through="none">
  <!-- declare variables -->
  <stx:variable name="itemnumber"/>
  <stx:variable name="price"/>
  <!-- match and output prices as columns delimited by | -->
  <stx:template match="/prices/pricerecord">
    <stx:process-children/>
    <stx:value-of select="$itemnumber"/>
<stx:text>|</stx:text>
    <stx:value-of select="$price"/>      <stx:text>
</stx:text>
  </stx:template>
  <stx:template match="itemnumber">
    <stx:assign name="itemnumber" select="."/>
  </stx:template>
  <stx:template match="price">
    <stx:assign name="price" select="."/>
  </stx:template>
</stx:transform>
```

This STX transform declares two temporary variables, `itemnumber` and `price`, and the following rules.

1. When an element that satisfies the XPath expression `/prices/pricerecord` is found, examine the child elements and generate output that contains the value of the `itemnumber` variable, a `|` character, the value of the price variable, and a newline.
2. When an <`itemnumber`> element is found, store the content of that element in the variable `itemnumber`.
3. When a <price> element is found, store the content of that element in the variable `price`.

Write the gpfdist Configuration File

The `gpfdist` configuration is specified as a YAML 1.1 document. It contains rules that `gpfdist` uses to select a transformation to apply when loading or extracting data.

This example `gpfdist` configuration contains the following items that are required for the *prices.xml* transformation scenario:

- the `config.yaml` file defining `TRANSFORMATIONS`
- the `input_transform.sh` wrapper script, referenced in the `config.yaml` file
- the `input_transform.stx` joost transformation, called from `input_transform.sh`

Aside from the ordinary YAML rules, such as starting the document with three dashes (`---`), a `gpfdist` configuration must conform to the following restrictions:

1. A `VERSION` setting must be present with the value `1.0.0.1`.
2. A `TRANSFORMATIONS` setting must be present and contain one or more mappings.
3. Each mapping in the `TRANSFORMATION` must contain:
   - a `TYPE` with the value 'input' or 'output'
   - a `COMMAND` indicating how the transformation is run.
4. Each mapping in the `TRANSFORMATION` can contain optional `CONTENT`, `SAFE`, and `STDERR` settings.

The following `gpfdist` configuration, called `config.yaml`, applies to the prices example. The initial indentation on each line is significant and reflects the hierarchical nature of the specification. The transformation name `prices_input` in the following example will be referenced later when creating the table in SQL.

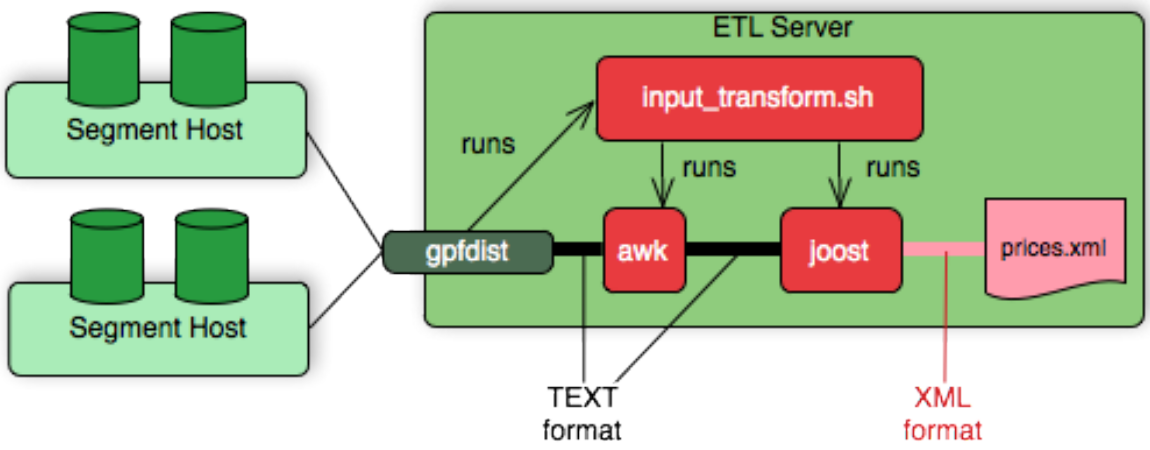
```
---
VERSION: 1.0.0.1
TRANSFORMATIONS:
  prices_input:
    TYPE:     input
    COMMAND:  /bin/bash input_transform.sh %filename%
```

The `COMMAND` setting uses a wrapper script called `input_transform.sh` with a `%filename%` placeholder. When `gpfdist` runs the `prices_input` transform, it invokes `input_transform.sh` with `/bin/bash` and replaces the `%filename%` placeholder with the path to the input file to transform. The wrapper script called `input_transform.sh` contains the logic to invoke the STX transformation and return the output.

If Joost is used, the Joost STX engine must be installed.

```
#!/bin/bash
# input_transform.sh - sample input transformation,
# demonstrating use of Java and Joost STX to convert XML into
# text to load into Greenplum Database.
# java arguments:
#   -jar joost.jar         joost STX engine
#   -nodecl                  don't generate a <?xml?> declaration
#   $1                       filename to process
#   input_transform.stx    the STX transformation
#
# the AWK step eliminates a blank line joost emits at the end
java \
    -jar joost.jar \
    -nodecl \
    $1 \
    input_transform.stx \
 | awk 'NF>0'
```

The `input_transform.sh` file uses the Joost STX engine with the AWK interpreter. The following diagram shows the process flow as `gpfdist` runs the transformation.



Transfer the Data

Create the target database tables with SQL statements based on the appropriate schema.

There are no special requirements for Greenplum Database tables that hold loaded data. In the prices example, the following command creates the `prices` table, where the data is to be loaded.

```
CREATE TABLE prices (
  itemnumber integer,
  price       decimal
)
DISTRIBUTED BY (itemnumber);
```

Next, use one of the following approaches to transform the data with `gpfdist`.

- `gpload` supports only input transformations, but in many cases is easier to implement.
- `gpfdist` with `INSERT INTO SELECT FROM` supports both input and output transformations, but exposes details that `gpload` automates for you.

Transforming with gpload

The Greenplum Database `gpload` utility orchestrates a data load operation using the `gpfdist` parallel file server and a YAML-formatted configuration file. `gpload` automates these tasks:

- Creates a readable external table in the database.
- Starts `gpfdist` instances with the configuration file that contains the transformation.
- Runs `INSERT INTO` *`table_name`* `SELECT FROM` *`external_table`* to load the data.
- Removes the external table definition.

Transforming data with `gpload` requires that the settings `TRANSFORM` and `TRANSFORM_CONFIG` appear in the `INPUT` section of the `gpload` control file.

For more information about the syntax and placement of these settings in the `gpload` control file, see the *Greenplum Database Reference Guide*.

- `TRANSFORM_CONFIG` specifies the name of the `gpfdist` configuration file.
- The `TRANSFORM` setting indicates the name of the transformation that is described in the file named in `TRANSFORM_CONFIG`.

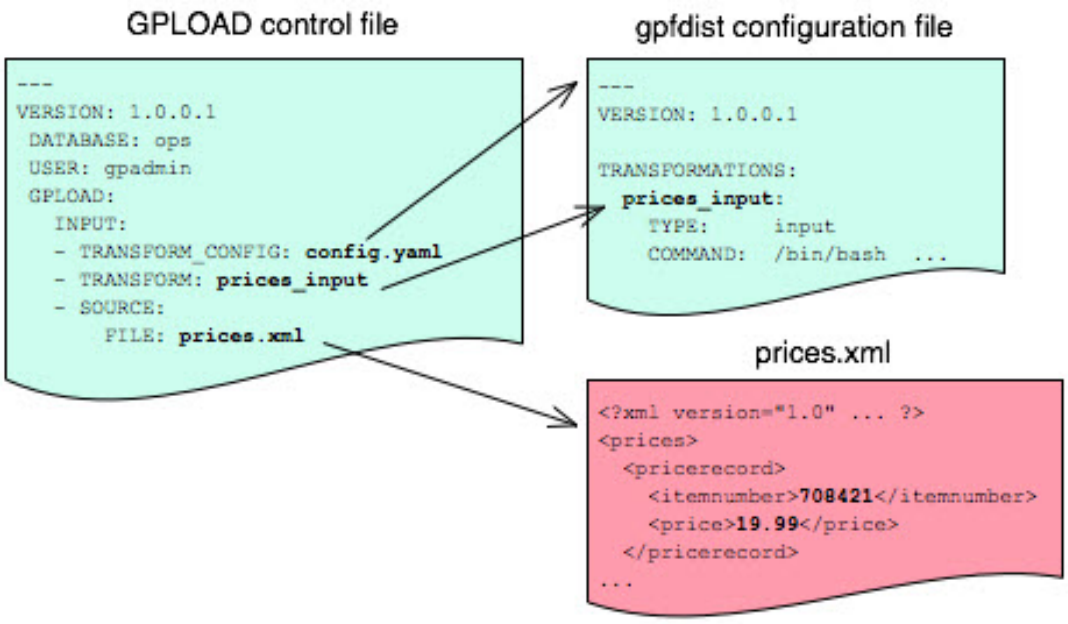
```
---
VERSION: 1.0.0.1
DATABASE: ops
```

```
USER: gpadmin
GPLOAD:
  INPUT:
    - TRANSFORM_CONFIG: config.yaml
    - TRANSFORM: prices_input
    - SOURCE:
        FILE: prices.xml
```

The transformation name must appear in two places: in the `TRANSFORM` setting of the `gpfdist` configuration file and in the `TRANSFORMATIONS` section of the file named in the `TRANSFORM_CONFIG` section.

In the `gpload` control file, the optional parameter `MAX_LINE_LENGTH` specifies the maximum length of a line in the XML transformation data that is passed to `gpload`.

The following diagram shows the relationships between the `gpload` control file, the `gpfdist` configuration file, and the XML data file.



Transforming with gpfdist and INSERT INTO SELECT FROM

With this load method, you perform each of the tasks that `gpload` automates. You start `gpfdist`, create an external table, load the data, and clean up by dropping the table and stopping `gpfdist`.

Specify the transformation in the `CREATE EXTERNAL TABLE` definition's `LOCATION` clause. For example, the transform is shown in bold in the following command. (Run `gpfdist` first, using the command `gpfdist -c config.yaml`).

```
CREATE READABLE EXTERNAL TABLE prices_readable (LIKE prices)
   LOCATION ('gpfdist://hostname:8080/prices.xml#transform=prices_input')
   FORMAT 'TEXT' (DELIMITER '|')
   LOG ERRORS SEGMENT REJECT LIMIT 10;
```

In the command above, change `hostname` to your hostname. `prices_input` comes from the `gpfdist` configuration file.

The following query then loads the data into the `prices` table.

```
INSERT INTO prices SELECT * FROM prices_readable;
```

Configuration File Format

The `gpfdist` configuration file uses the YAML 1.1 document format and implements a schema for defining the transformation parameters. The configuration file must be a valid YAML document.

The `gpfdist` program processes the document in order and uses indentation (spaces) to determine the document hierarchy and relationships of the sections to one another. The use of white space is significant. Do not use white space for formatting and do not use tabs.

The following is the basic structure of a configuration file.

```
---
VERSION:   1.0.0.1
TRANSFORMATIONS:
  transformation_name1:
    TYPE:      input | output
    COMMAND:   command
    CONTENT:   data | paths
    SAFE:      posix-regex
    STDERR:    server | console
  transformation_name2:
    TYPE:      input | output
    COMMAND:   command
...
```

VERSION

Required. The version of the `gpfdist` configuration file schema. The current version is 1.0.0.1.

TRANSFORMATIONS

Required. Begins the transformation specification section. A configuration file must have at least one transformation. When `gpfdist` receives a transformation request, it looks in this section for an entry with the matching transformation name.

TYPE

Required. Specifies the direction of transformation. Values are `input` or `output`.

- `input`: `gpfdist` treats the standard output of the transformation process as a stream of records to load into Greenplum Database.
- `output` : `gpfdist` treats the standard input of the transformation process as a stream of records from Greenplum Database to transform and write to the appropriate output.

COMMAND

Required. Specifies the command `gpfdist` will execute to perform the transformation.

For input transformations, `gpfdist` invokes the command specified in the `CONTENT` setting. The command is expected to open the underlying file(s) as appropriate and produce one line of `TEXT` for each row to load into Greenplum Database. The input transform determines whether the entire content should be converted to one row or to multiple rows.

For output transformations, `gpfdist` invokes this command as specified in the `CONTENT` setting. The output command is expected to open and write to the underlying file(s) as appropriate. The output transformation determines the final placement of the converted output.

CONTENT

Optional. The values are `data` and `paths`. The default value is `data`.

- When `CONTENT` specifies `data`, the text `%filename%` in the `COMMAND` section is replaced by the path to the file to read or write.
- When `CONTENT` specifies `paths`, the text `%filename%` in the `COMMAND` section is replaced by the path to the temporary file that contains the list of files to read or write.

The following is an example of a `COMMAND` section showing the text `%filename%` that is replaced.

```
COMMAND: /bin/bash input_transform.sh %filename%
```

SAFE

Optional. A `POSIX` regular expression that the paths must match to be passed to the transformation. Specify `SAFE` when there is a concern about injection or improper interpretation of paths passed to the command. The default is no restriction on paths.

STDERR

Optional. The values are `server` and `console`.

This setting specifies how to handle standard error output from the transformation. The default, `server`, specifies that `gpfdist` will capture the standard error output from the transformation in a temporary file and send the first 8k of that file to Greenplum Database as an error message. The error message will appear as an SQL error. `Console` specifies that `gpfdist` does not redirect or transmit the standard error output from the transformation.

XML Transformation Examples

The following examples demonstrate the complete process for different types of XML data and STX transformations. Files and detailed instructions associated with these examples are in the GitHub repo `github.com://greenplum-db/gpdb` in the *`gpMgmt/demo/gpfdist_transform`* directory. Read the README file in the *Before You Begin* section before you run the examples. The README file explains how to download the example data file used in the examples.

Command-based External Web Tables

The output of a shell command or script defines command-based web table data. Specify the command in the `EXECUTE` clause of `CREATE EXTERNAL WEB TABLE`. The data is current as of the time the command runs. The `EXECUTE` clause runs the shell command or script on the specified master, and/or segment host or hosts. The command or script must reside on the hosts corresponding to the host(s) defined in the `EXECUTE` clause.

By default, the command is run on segment hosts when active segments have output rows to process. For example, if each segment host runs four primary segment instances that have output rows to process, the command runs four times per segment host. You can optionally limit the number of segment instances that execute the web table command. All segments included in the web table definition in the `ON` clause run the command in parallel.

The command that you specify in the external table definition executes from the database and cannot access environment variables from `.bashrc` or `.profile`. Set environment variables in the `EXECUTE` clause. For example:

```
=# CREATE EXTERNAL WEB TABLE output (output text)
    EXECUTE 'PATH=/home/gpadmin/programs; export PATH; myprogram.sh'
    FORMAT 'TEXT';
```

Scripts must be executable by the `gpadmin` user and reside in the same location on the master or segment hosts.

The following command defines a web table that runs a script. The script runs on each segment host where a segment has output rows to process.

```
=# CREATE EXTERNAL WEB TABLE log_output
    (linenum int, message text)
    EXECUTE '/var/load_scripts/get_log_data.sh' ON HOST
    FORMAT 'TEXT' (DELIMITER '|');
```

IRS MeF XML Files (In demo Directory)

This example demonstrates loading a sample IRS Modernized eFile tax return using a Joost STX transformation. The data is in the form of a complex XML file.

The U.S. Internal Revenue Service (IRS) made a significant commitment to XML and specifies its use in its Modernized e-File (MeF) system. In MeF, each tax return is an XML document with a deep hierarchical structure that closely reflects the particular form of the underlying tax code.

XML, XML Schema and stylesheets play a role in their data representation and business workflow. The actual XML data is extracted from a ZIP file attached to a MIME "transmission file" message. For more information about MeF, see *Modernized e-File (Overview)* on the IRS web site.

The sample XML document, *RET990EZ_2006.xml*, is about 350KB in size with two elements:

- ReturnHeader
- ReturnData

The <ReturnHeader> element contains general details about the tax return such as the taxpayer's name, the tax year of the return, and the preparer. The <ReturnData> element contains multiple sections with specific details about the tax return and associated schedules.

The following is an abridged sample of the XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<Return returnVersion="2006v2.0"
   xmlns="https://www.irs.gov/efile"
   xmlns:efile="https://www.irs.gov/efile"
   xsi:schemaLocation="https://www.irs.gov/efile"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <ReturnHeader binaryAttachmentCount="1">
     <ReturnId>AAAAAAAAAAAAAAAAAAAA</ReturnId>
     <Timestamp>1999-05-30T12:01:01+05:01</Timestamp>
     <ReturnType>990EZ</ReturnType>
     <TaxPeriodBeginDate>2005-01-01</TaxPeriodBeginDate>
     <TaxPeriodEndDate>2005-12-31</TaxPeriodEndDate>
     <Filer>
       <EIN>011248772</EIN>
       ... more data ...
     </Filer>
     <Preparer>
       <Name>Percy Polar</Name>
       ... more data ...
     </Preparer>
     <TaxYear>2005</TaxYear>
   </ReturnHeader>
   ... more data ..
```

The goal is to import all the data into a Greenplum database. First, convert the XML document into text with newlines "escaped", with two columns: `ReturnId` and a single column on the end for the entire MeF tax return. For example:

```
AAAAAAAAAAAAAAAAAAAA|<Return returnVersion="2006v2.0"...
```

Load the data into Greenplum Database.

WITSML™ Files (In demo Directory)

This example demonstrates loading sample data describing an oil rig using a Joost STX transformation. The data is in the form of a complex XML file downloaded from energistics.org.

The Wellsite Information Transfer Standard Markup Language (WITSML™) is an oil industry initiative to provide open, non-proprietary, standard interfaces for technology and software to share information among oil companies, service companies, drilling contractors, application vendors, and regulatory agencies. For more information about WITSML™, see *http://www.energistics.org/*.

The oil rig information consists of a top level `<rigs>` element with multiple child elements such as `<documentInfo>, <rig>`, and so on. The following excerpt from the file shows the type of information in the `<rig>` tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="../stylesheets/rig.xsl" type="text/xsl"
 media="screen"?>
<rigs
 xmlns="http://www.energistics.org/schemas/131"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.energistics.org/schemas/131 ../obj_rig.xsd"
 version="1.3.1.1">
 <documentInfo>
 ... misc data ...
 </documentInfo>
 <rig uidWell="W-12" uidWellbore="B-01" uid="xr31">
     <nameWell>6507/7-A-42</nameWell>
     <nameWellbore>A-42</nameWellbore>
     <name>Deep Drill #5</name>
     <owner>Deep Drilling Co.</owner>
     <typeRig>floater</typeRig>
     <manufacturer>Fitsui Engineering</manufacturer>
     <yearEntService>1980</yearEntService>
     <classRig>ABS Class A1 M CSDU AMS ACCU</classRig>
     <approvals>DNV</approvals>
 ... more data ...
```

The goal is to import the information for this rig into Greenplum Database.

The sample document, *rig.xml*, is about 11KB in size. The input does not contain tabs so the relevant information can be converted into records delimited with a pipe (|).

```
W-12|6507/7-A-42|xr31|Deep Drill #5|Deep Drilling Co.|John Doe|
John.Doe@example.com|
```

With the columns:

- `well_uid text`, -- e.g. W-12
- `well_name text`, -- e.g. 6507/7-A-42
- `rig_uid text`, -- e.g. xr31
- `rig_name text`, -- e.g. Deep Drill #5
- `rig_owner text`, -- e.g. Deep Drilling Co.
- `rig_contact text`, -- e.g. John Doe
- `rig_email text`, -- e.g. John.Doe@example.com
- `doc xml`

Then, load the data into Greenplum Database.

Loading Data with COPY

`COPY FROM` copies data from a file or standard input into a table and appends the data to the table contents. `COPY` is non-parallel: data is loaded in a single process using the Greenplum master instance. Using `COPY` is only recommended for very small data files.

The `COPY` source file must be accessible to the `postgres` process on the master host. Specify the `COPY` source file name relative to the data directory on the master host, or specify an absolute path.

Greenplum copies data from `STDIN` or `STDOUT` using the connection between the client and the master server.

Loading From a File

The `COPY` command asks the `postgres` backend to open the specified file, read it and append it to the table. In order to be able to read the file, the backend needs to have read permissions on the file, and the file name must be specified using an absolute path on the master host, or a relative path to the master data directory.

```
COPY table_name FROM /path/to/filename;
```

Loading From STDIN

To avoid the problem of copying the data file to the master host before loading the data, `COPY FROM STDIN` uses the Standard Input channel and feeds data directly into the `postgres` backend. After the `COPY FROM STDIN` command started, the backend will accept lines of data until a single line only contains a backslash-period (\.).

```
COPY table_name FROM STDIN;
```

Loading Data Using \copy in psql

Do not confuse the psql `\copy` command with the `COPY` SQL command. The `\copy` invokes a regular `COPY FROM STDIN` and sends the data from the psql client to the backend. Therefore any file must reside on the host where the psql client runs, and must be accessible to the user which runs the client.

To avoid the problem of copying the data file to the master host before loading the data, `COPY FROM STDIN` uses the Standard Input channel and feeds data directly into the `postgres` backend. After the `COPY FROM STDIN` command started, the backend will accept lines of data until a single line only contains a backslash-period (\.). psql is wrapping all of this into the handy `\copy` command.

```
\copy table_name FROM filename;
```

Input Format

`COPY FROM` accepts a `FORMAT` parameter, which specifies the format of the input data. The possible values are `TEXT`, `CSV` (Comma Separated Values), and `BINARY`.

```
COPY table_name FROM /path/to/filename WITH (FORMAT csv);
```

The `FORMAT csv` will read comma-separated values. The `FORMAT text` by default uses tabulators to separate the values, the `DELIMITER` option specifies a different character as value delimiter.

```
COPY table_name FROM /path/to/filename WITH (FORMAT text, DELIMITER '|');
```

By default, the default client encoding is used, this can be changed with the `ENCODING` option. This is useful if data is coming from another operating system.

```
COPY table_name FROM /path/to/filename WITH (ENCODING 'latin1');
```

Running COPY in Single Row Error Isolation Mode

By default, `COPY` stops an operation at the first error: if the data contains an error, the operation fails and no data loads. If you run `COPY FROM` in *single row error isolation mode*, Greenplum skips rows that contain format errors and loads properly formatted rows. Single row error isolation mode applies only to rows in the input file that contain format errors. If the data contains a constraint error such as violation of a `NOT NULL`, `CHECK`, or `UNIQUE` constraint, the operation fails and no data loads.

Specifying `SEGMENT REJECT LIMIT` runs the `COPY` operation in single row error isolation mode. Specify the acceptable number of error rows on each segment, after which the entire `COPY FROM` operation fails and no rows load. The error row count is for each Greenplum Database segment, not for the entire load operation.

If the `COPY` operation does not reach the error limit, Greenplum loads all correctly-formatted rows and discards the error rows. Use the `LOG ERRORS` clause to capture data formatting errors internally in Greenplum Database. For example:

```
=> COPY country FROM '/data/gpdb/country_data'
   WITH DELIMITER '|' LOG ERRORS
   SEGMENT REJECT LIMIT 10 ROWS;
```

See *Viewing Bad Rows in the Error Log* for information about investigating error rows.

Optimizing Data Load and Query Performance

Use the following tips to help optimize your data load and subsequent query performance.

- Drop indexes before loading data into existing tables.

  Creating an index on pre-existing data is faster than updating it incrementally as each row is loaded. You can temporarily increase the `maintenance_work_mem` server configuration parameter to help speed up `CREATE INDEX` commands, though load performance is affected. Drop and recreate indexes only when there are no active users on the system.
- Create indexes last when loading data into new tables. Create the table, load the data, and create any required indexes.
- Run `ANALYZE` after loading data. If you significantly altered the data in a table, run `ANALYZE` or `VACUUM ANALYZE` to update table statistics for the query optimizer. Current statistics ensure that the optimizer makes the best decisions during query planning and avoids poor performance due to inaccurate or nonexistent statistics.
- Run `VACUUM` after load errors. If the load operation does not run in single row error isolation mode, the operation stops at the first error. The target table contains the rows loaded before the error occurred. You cannot access these rows, but they occupy disk space. Use the `VACUUM` command to recover the wasted space.

Unloading Data from Greenplum Database

A writable external table allows you to select rows from other database tables and output the rows to files, named pipes, to applications, or as output targets for Greenplum parallel MapReduce calculations. You can define file-based and web-based writable external tables.

This topic describes how to unload data from Greenplum Database using parallel unload (writable external tables) and non-parallel unload (COPY).

Defining a File-Based Writable External Table

Writable external tables that output data to files can use the Greenplum parallel file server program, gpfdist, or the Greenplum Platform Extension Framework (PXF), Greenplum's interface to Hadoop.

Use the CREATE WRITABLE EXTERNAL TABLE command to define the external table and specify the location and format of the output files. See *Using the Greenplum Parallel File Server (gpfdist)* for instructions on setting up gpfdist for use with an external table and *Accessing External Data with PXF* for instructions on setting up PXF for use with an external table

- With a writable external table using the gpfdist protocol, the Greenplum segments send their data to gpfdist, which writes the data to the named file. gpfdist must run on a host that the Greenplum segments can access over the network. gpfdist points to a file location on the output host and writes data received from the Greenplum segments to the file. To divide the output data among multiple files, list multiple gpfdist URIs in your writable external table definition.
- A writable external web table sends data to an application as a stream of data. For example, unload data from Greenplum Database and send it to an application that connects to another database or ETL tool to load the data elsewhere. Writable external web tables use the EXECUTE clause to specify a shell command, script, or application to run on the segment hosts and accept an input stream of data. See *Defining a Command-Based Writable External Web Table* for more information about using EXECUTE commands in a writable external table definition.

You can optionally declare a distribution policy for your writable external tables. By default, writable external tables use a random distribution policy. If the source table you are exporting data from has a hash distribution policy, defining the same distribution key column(s) for the writable external table improves unload performance by eliminating the requirement to move rows over the interconnect. If you unload data from a particular table, you can use the LIKE clause to copy the column definitions and distribution policy from the source table.

Example 1—Greenplum file server (gpfdist)

```
=# CREATE WRITABLE EXTERNAL TABLE unload_expenses
   ( LIKE expenses )
   LOCATION ('gpfdist://etlhost-1:8081/expenses1.out',
             'gpfdist://etlhost-2:8081/expenses2.out')
 FORMAT 'TEXT' (DELIMITER ',')
 DISTRIBUTED BY (exp_id);
```

Example 2—Hadoop file server (pxf)

```
=# CREATE WRITABLE EXTERNAL TABLE unload_expenses
   ( LIKE expenses )
   LOCATION ('pxf://dir/path?PROFILE=hdfs:text')
 FORMAT 'TEXT' (DELIMITER ',')
 DISTRIBUTED BY (exp_id);
```

You specify an HDFS directory for a writable external table that you create with the pxf protocol.

Defining a Command-Based Writable External Web Table

You can define writable external web tables to send output rows to an application or script. The application must accept an input stream, reside in the same location on all of the Greenplum segment hosts, and be executable by the gpadmin user. All segments in the Greenplum system run the application or script, whether or not a segment has output rows to process.

Use `CREATE WRITABLE EXTERNAL WEB TABLE` to define the external table and specify the application or script to run on the segment hosts. Commands execute from within the database and cannot access environment variables (such as `$PATH`). Set environment variables in the `EXECUTE` clause of your writable external table definition. For example:

```
=# CREATE WRITABLE EXTERNAL WEB TABLE output (output text)
    EXECUTE 'export PATH=$PATH:/home/gpadmin
            /programs;
    myprogram.sh'
    FORMAT 'TEXT'
    DISTRIBUTED RANDOMLY;
```

The following Greenplum Database variables are available for use in OS commands executed by a web or writable external table. Set these variables as environment variables in the shell that executes the command(s). They can be used to identify a set of requests made by an external table statement across the Greenplum Database array of hosts and segment instances.

Table 58: External Table EXECUTE Variables

Variable	Description
$GP_CID	Command count of the transaction executing the external table statement.
$GP_DATABASE	The database in which the external table definition resides.
$GP_DATE	The date on which the external table command ran.
$GP_MASTER_HOST	The host name of the Greenplum master host from which the external table statement was dispatched.
$GP_MASTER_PORT	The port number of the Greenplum master instance from which the external table statement was dispatched.
$GP_QUERY_STRING	The SQL command (DML or SQL query) executed by Greenplum Database.
$GP_SEG_DATADIR	The location of the data directory of the segment instance executing the external table command.
$GP_SEG_PG_CONF	The location of the `postgresql.conf` file of the segment instance executing the external table command.
$GP_SEG_PORT	The port number of the segment instance executing the external table command.
$GP_SEGMENT_COUNT	The total number of primary segment instances in the Greenplum Database system.
$GP_SEGMENT_ID	The ID number of the segment instance executing the external table command (same as `dbid` in `gp_segment_configuration`).
$GP_SESSION_ID	The database session identifier number associated with the external table statement.
$GP_SN	Serial number of the external table scan node in the query plan of the external table statement.
$GP_TIME	The time the external table command was executed.
$GP_USER	The database user executing the external table statement.
$GP_XID	The transaction ID of the external table statement.

Disabling EXECUTE for Web or Writable External Tables

There is a security risk associated with allowing external tables to execute OS commands or scripts. To disable the use of `EXECUTE` in web and writable external table definitions, set the `gp_external_enable_exec server` configuration parameter to off in your master `postgresql.conf` file:

```
gp_external_enable_exec = off
```

Unloading Data Using a Writable External Table

Writable external tables allow only `INSERT` operations. You must grant `INSERT` permission on a table to enable access to users who are not the table owner or a superuser. For example:

```
GRANT INSERT ON writable_ext_table TO admin;
```

To unload data using a writable external table, select the data from the source table(s) and insert it into the writable external table. The resulting rows are output to the writable external table. For example:

```
INSERT INTO writable_ext_table SELECT * FROM regular_table;
```

Unloading Data Using COPY

`COPY TO` copies data from a table to a file (or standard input) on the Greenplum master host using a single process on the Greenplum master instance. Use `COPY` to output a table's entire contents, or filter the output using a `SELECT` statement. For example:

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%')
TO '/home/gpadmin/a_list_countries.out';
```

Formatting Data Files

When you use the Greenplum tools for loading and unloading data, you must specify how your data is formatted. `COPY`, `CREATE EXTERNAL TABLE`, and `gpload` have clauses that allow you to specify how your data is formatted. Data can be delimited text (`TEXT`) or comma separated values (`CSV`) format. External data must be formatted correctly to be read by Greenplum Database. This topic explains the format of data files expected by Greenplum Database.

Formatting Rows

Greenplum Database expects rows of data to be separated by the `LF` character (Line feed, `0x0A`), `CR` (Carriage return, `0x0D`), or `CR` followed by `LF` (`CR+LF`, `0x0D 0x0A`). `LF` is the standard newline representation on UNIX or UNIX-like operating systems. Operating systems such as Windows or Mac OS X use `CR` or `CR+LF`. All of these representations of a newline are supported by Greenplum Database as a row delimiter. For more information, see *Importing and Exporting Fixed Width Data*.

Formatting Columns

The default column or field delimiter is the horizontal `TAB` character (`0x09`) for text files and the comma character (`0x2C`) for CSV files. You can declare a single character delimiter using the `DELIMITER` clause of `COPY`, `CREATE EXTERNAL TABLE` or `gpload` when you define your data format. The delimiter

character must appear between any two data value fields. Do not place a delimiter at the beginning or end of a row. For example, if the pipe character (|) is your delimiter:

```
data value 1|data value 2|data value 3
```

The following command shows the use of the pipe character as a column delimiter:

```
=# CREATE EXTERNAL TABLE ext_table (name text, date date)
LOCATION ('gpfdist://<hostname>/filename.txt)
FORMAT 'TEXT' (DELIMITER '|');
```

Representing NULL Values

`NULL` represents an unknown piece of data in a column or field. Within your data files you can designate a string to represent null values. The default string is `\N` (backslash-N) in `TEXT` mode, or an empty value with no quotations in `CSV` mode. You can also declare a different string using the `NULL` clause of `COPY`, `CREATE EXTERNAL TABLE` or `gpload` when defining your data format. For example, you can use an empty string if you do not want to distinguish nulls from empty strings. When using the Greenplum Database loading tools, any data item that matches the designated null string is considered a null value.

Escaping

There are two reserved characters that have special meaning to Greenplum Database:

- The designated delimiter character separates columns or fields in the data file.
- The newline character designates a new row in the data file.

If your data contains either of these characters, you must escape the character so that Greenplum treats it as data and not as a field separator or new row. By default, the escape character is a \ (backslash) for text-formatted files and a double quote (") for csv-formatted files.

Escaping in Text Formatted Files

By default, the escape character is a \ (backslash) for text-formatted files. You can declare a different escape character in the `ESCAPE` clause of `COPY`, `CREATE EXTERNAL TABLE` or `gpload`. If your escape character appears in your data, use it to escape itself.

For example, suppose you have a table with three columns and you want to load the following three fields:

- `backslash = \`
- `vertical bar = |`
- `exclamation point = !`

Your designated delimiter character is `|` (pipe character), and your designated escape character is `\` (backslash). The formatted row in your data file looks like this:

```
backslash = \\ | vertical bar = \| | exclamation point = !
```

Notice how the backslash character that is part of the data is escaped with another backslash character, and the pipe character that is part of the data is escaped with a backslash character.

You can use the escape character to escape octal and hexadecimal sequences. The escaped value is converted to the equivalent character when loaded into Greenplum Database. For example, to load the ampersand character (`&`), use the escape character to escape its equivalent hexadecimal (`\0x26`) or octal (`\046`) representation.

You can disable escaping in TEXT-formatted files using the ESCAPE clause of COPY, CREATE EXTERNAL TABLE or gpload as follows:

```
ESCAPE 'OFF'
```

This is useful for input data that contains many backslash characters, such as web log data.

Escaping in CSV Formatted Files

By default, the escape character is a " (double quote) for CSV-formatted files. If you want to use a different escape character, use the ESCAPE clause of COPY, CREATE EXTERNAL TABLE or gpload to declare a different escape character. In cases where your selected escape character is present in your data, you can use it to escape itself.

For example, suppose you have a table with three columns and you want to load the following three fields:

- Free trip to A,B
- 5.89
- Special rate "1.79"

Your designated delimiter character is , (comma), and your designated escape character is " (double quote). The formatted row in your data file looks like this:

```
"Free trip to A,B","5.89","Special rate ""1.79"""
```

The data value with a comma character that is part of the data is enclosed in double quotes. The double quotes that are part of the data are escaped with a double quote even though the field value is enclosed in double quotes.

Embedding the entire field inside a set of double quotes guarantees preservation of leading and trailing whitespace characters:

"Free trip to A,B ","5.89 ","Special rate ""1.79"" "

> **Note:** In CSV mode, all characters are significant. A quoted value surrounded by white space, or any characters other than DELIMITER, includes those characters. This can cause errors if you import data from a system that pads CSV lines with white space to some fixed width. In this case, preprocess the CSV file to remove the trailing white space before importing the data into Greenplum Database.

Character Encoding

Character encoding systems consist of a code that pairs each character from a character set with something else, such as a sequence of numbers or octets, to facilitate data stransmission and storage. Greenplum Database supports a variety of character sets, including single-byte character sets such as the ISO 8859 series and multiple-byte character sets such as EUC (Extended UNIX Code), UTF-8, and Mule internal code. The server-side character set is defined during database initialization, UTF-8 is the default and can be changed. Clients can use all supported character sets transparently, but a few are not supported for use within the server as a server-side encoding. When loading or inserting data into Greenplum Database, Greenplum transparently converts the data from the specified client encoding into the server encoding. When sending data back to the client, Greenplum converts the data from the server character encoding into the specified client encoding.

Data files must be in a character encoding recognized by Greenplum Database. See the *Greenplum Database Reference Guide* for the supported character sets. Data files that contain invalid or unsupported encoding sequences encounter errors when loading into Greenplum Database.

Note: On data files generated on a Microsoft Windows operating system, run the `dos2unix` system command to remove any Windows-only characters before loading into Greenplum Database.

Note: If you *change* the `ENCODING` value in an existing `gpload` control file, you must manually drop any external tables that were creating using the previous `ENCODING` configuration. `gpload` does not drop and recreate external tables to use the new `ENCODING` if `REUSE_TABLES` is set to `true`.

Changing the Client-Side Character Encoding

The client-side character encoding can be changed for a session by setting the server configuration parameter `client_encoding`

```
SET client_encoding TO 'latin1';
```

Change the client-side character encoding back to the default value:

```
RESET client_encoding;
```

Show the current client-side character encoding setting:

```
SHOW client_encoding;
```

Example Custom Data Access Protocol

The following is the API for the Greenplum Database custom data access protocol. The example protocol implementation *gpextprotocal.c* is written in C and shows how the API can be used. For information about accessing a custom data access protocol, see *Using a Custom Protocol*.

```
/* ---- Read/Write function API ------*/
CALLED_AS_EXTPROTOCOL(fcinfo)
EXTPROTOCOL_GET_URL(fcinfo)(fcinfo)
EXTPROTOCOL_GET_DATABUF(fcinfo)
EXTPROTOCOL_GET_DATALEN(fcinfo)
EXTPROTOCOL_GET_SCANQUALS(fcinfo)
EXTPROTOCOL_GET_USER_CTX(fcinfo)
EXTPROTOCOL_IS_LAST_CALL(fcinfo)
EXTPROTOCOL_SET_LAST_CALL(fcinfo)
EXTPROTOCOL_SET_USER_CTX(fcinfo, p)

/* ------ Validator function API ------*/
CALLED_AS_EXTPROTOCOL_VALIDATOR(fcinfo)
EXTPROTOCOL_VALIDATOR_GET_URL_LIST(fcinfo)
EXTPROTOCOL_VALIDATOR_GET_NUM_URLS(fcinfo)
EXTPROTOCOL_VALIDATOR_GET_NTH_URL(fcinfo, n)
EXTPROTOCOL_VALIDATOR_GET_DIRECTION(fcinfo)
```

Notes

The protocol corresponds to the example described in *Using a Custom Protocol*. The source code file name and shared object are `gpextprotocol.c` and `gpextprotocol.so`.

The protocol has the following properties:

- The name defined for the protocol is `myprot`.
- The protocol has the following simple form: the protocol name and a path, separated by `://`.

  `myprot://` `path`
- Three functions are implemented:

- `myprot_import()` a read function
- `myprot_export()` a write function
- `myprot_validate_urls()` a validation function

These functions are referenced in the `CREATE PROTOCOL` statement when the protocol is created and declared in the database.

The example implementation *gpextprotocal.c* uses `fopen()` and `fread()` to simulate a simple protocol that reads local files. In practice, however, the protocol would implement functionality such as a remote connection to some process over the network.

Installing the External Table Protocol

To use the example external table protocol, you use the C compiler `cc` to compile and link the source code to create a shared object that can be dynamically loaded by Greenplum Database. The commands to compile and link the source code on a Linux system are similar to this:

```
cc -fpic -c gpextprotocal.c cc -shared -o gpextprotocal.so gpextprotocal.o
```

The option `-fpic` specifies creating position-independent code (PIC) and the `-c` option compiles the source code without linking and creates an object file. The object file needs to be created as position-independent code (PIC) so that it can be loaded at any arbitrary location in memory by Greenplum Database.

The flag `-shared` specifies creating a shared object (shared library) and the `-o` option specifies the shared object file name `gpextprotocal.so`. Refer to the GCC manual for more information on the `cc` options.

The header files that are declared as include files in `gpextprotocal.c` are located in subdirectories of `$GPHOME/include/postgresql/.`

For more information on compiling and linking dynamically-loaded functions and examples of compiling C source code to create a shared library on other operating systems, see the PostgreSQL documentation at *https://www.postgresql.org/docs/9.4/xfunc-c.html#DFUNC* .

The manual pages for the C compiler `cc` and the link editor `ld` for your operating system also contain information on compiling and linking source code on your system.

The compiled code (shared object file) for the custom protocol must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files. It is recommended to locate shared libraries either relative to `$libdir` (which is located at `$GPHOME/lib`) or through the dynamic library path (set by the `dynamic_library_path` server configuration parameter) on all master segment instances in the Greenplum Database array. You can use the Greenplum Database utilities `gpssh` and `gpscp` to update segments.

gpextprotocal.c

```
#include "postgres.h"
#include "fmgr.h"
#include "funcapi.h"
#include "access/extprotocol.h"
#include "catalog/pg_proc.h"
#include "utils/array.h"
#include "utils/builtins.h"
#include "utils/memutils.h"

/* Our chosen URI format. We can change it however needed */
typedef struct DemoUri
{
   char     *protocol;
```

```
   char     *path;
}  DemoUri;
static DemoUri *ParseDemoUri(const char *uri_str);
static void FreeDemoUri(DemoUri* uri);

/* Do the module magic dance */
PG_MODULE_MAGIC;
PG_FUNCTION_INFO_V1(demoprot_export);
PG_FUNCTION_INFO_V1(demoprot_import);
PG_FUNCTION_INFO_V1(demoprot_validate_urls);

Datum demoprot_export(PG_FUNCTION_ARGS);
Datum demoprot_import(PG_FUNCTION_ARGS);
Datum demoprot_validate_urls(PG_FUNCTION_ARGS);

/* A user context that persists across calls. Can be
declared in any other way */
typedef struct {
  char    *url;
  char    *filename;
  FILE    *file;
} extprotocol_t;
/*
* The read function - Import data into GPDB.
*/
Datum
myprot_import(PG_FUNCTION_ARGS)
{
  extprotocol_t   *myData;
  char            *data;
  int             datlen;
  size_t          nread = 0;

  /* Must be called via the external table format manager */
  if (!CALLED_AS_EXTPROTOCOL(fcinfo))
    elog(ERROR, "myprot_import: not called by external
       protocol manager");

  /* Get our internal description of the protocol */
  myData = (extprotocol_t *) EXTPROTOCOL_GET_USER_CTX(fcinfo);

  if(EXTPROTOCOL_IS_LAST_CALL(fcinfo))
  {
    /* we're done receiving data. close our connection */
    if(myData && myData->file)
      if(fclose(myData->file))
        ereport(ERROR,
          (errcode_for_file_access(),
           errmsg("could not close file \"%s\": %m",
               myData->filename)));

    PG_RETURN_INT32(0);
  }

  if (myData == NULL)
  {
    /* first call. do any desired init */

    const char    *p_name = "myprot";
    DemoUri       *parsed_url;
    char          *url = EXTPROTOCOL_GET_URL(fcinfo);
    myData        = palloc(sizeof(extprotocol_t));

    myData->url   = pstrdup(url);
```

```
    parsed_url    = ParseDemoUri(myData->url);
    myData->filename = pstrdup(parsed_url->path);

    if(strcasecmp(parsed_url->protocol, p_name) != 0)
      elog(ERROR, "internal error: myprot called with a
          different protocol (%s)",
            parsed_url->protocol);

    FreeDemoUri(parsed_url);

    /* open the destination file (or connect to remote server in
       other cases) */
    myData->file = fopen(myData->filename, "r");

    if (myData->file == NULL)
      ereport(ERROR,
          (errcode_for_file_access(),
           errmsg("myprot_import: could not open file \"%s\"
             for reading: %m",
             myData->filename),
           errOmitLocation(true)));

    EXTPROTOCOL_SET_USER_CTX(fcinfo, myData);
  }
  /* ========================================
   *          DO THE IMPORT
   * ======================================== */
  data    = EXTPROTOCOL_GET_DATABUF(fcinfo);
  datlen  = EXTPROTOCOL_GET_DATALEN(fcinfo);

  /* read some bytes (with fread in this example, but normally
     in some other method over the network) */
  if(datlen > 0)
  {
    nread = fread(data, 1, datlen, myData->file);
    if (ferror(myData->file))
      ereport(ERROR,
        (errcode_for_file_access(),
          errmsg("myprot_import: could not write to file
            \"%s\": %m",
            myData->filename)));
  }
  PG_RETURN_INT32((int)nread);
}
/*
 * Write function - Export data out of GPDB
 */
Datum
myprot_export(PG_FUNCTION_ARGS)
{
  extprotocol_t  *myData;
  char           *data;
  int            datlen;
  size_t         wrote = 0;

  /* Must be called via the external table format manager */
  if (!CALLED_AS_EXTPROTOCOL(fcinfo))
    elog(ERROR, "myprot_export: not called by external
       protocol manager");

  /* Get our internal description of the protocol */
  myData = (extprotocol_t *) EXTPROTOCOL_GET_USER_CTX(fcinfo);
  if(EXTPROTOCOL_IS_LAST_CALL(fcinfo))
  {
```

```
    /* we're done sending data. close our connection */
    if(myData && myData->file)
      if(fclose(myData->file))
        ereport(ERROR,
            (errcode_for_file_access(),
              errmsg("could not close file \"%s\": %m",
                 myData->filename)));

    PG_RETURN_INT32(0);
  }
  if (myData == NULL)
  {
    /* first call. do any desired init */
    const char *p_name = "myprot";
    DemoUri    *parsed_url;
    char       *url = EXTPROTOCOL_GET_URL(fcinfo);

    myData           = palloc(sizeof(extprotocol_t));

    myData->url      = pstrdup(url);
    parsed_url       = ParseDemoUri(myData->url);
    myData->filename = pstrdup(parsed_url->path);

    if(strcasecmp(parsed_url->protocol, p_name) != 0)
      elog(ERROR, "internal error: myprot called with a
         different protocol (%s)",
         parsed_url->protocol);

    FreeDemoUri(parsed_url);

    /* open the destination file (or connect to remote server in
    other cases) */
    myData->file = fopen(myData->filename, "a");
    if (myData->file == NULL)
      ereport(ERROR,
        (errcode_for_file_access(),
           errmsg("myprot_export: could not open file \"%s\"
             for writing: %m",
             myData->filename),
         errOmitLocation(true)));

    EXTPROTOCOL_SET_USER_CTX(fcinfo, myData);
  }
  /* ========================================
   *      DO THE EXPORT
   * ======================================== */
  data   = EXTPROTOCOL_GET_DATABUF(fcinfo);
  datlen   = EXTPROTOCOL_GET_DATALEN(fcinfo);

  if(datlen > 0)
  {
    wrote = fwrite(data, 1, datlen, myData->file);

    if (ferror(myData->file))
      ereport(ERROR,
        (errcode_for_file_access(),
         errmsg("myprot_import: could not read from file
            \"%s\": %m",
            myData->filename)));
  }
  PG_RETURN_INT32((int)wrote);
}
Datum
myprot_validate_urls(PG_FUNCTION_ARGS)
```

```
{
  List         *urls;
  int          nurls;
  int          i;
  ValidatorDirection  direction;

  /* Must be called via the external table format manager */
  if (!CALLED_AS_EXTPROTOCOL_VALIDATOR(fcinfo))
    elog(ERROR, "myprot_validate_urls: not called by external
       protocol manager");

  nurls       = EXTPROTOCOL_VALIDATOR_GET_NUM_URLS(fcinfo);
  urls        = EXTPROTOCOL_VALIDATOR_GET_URL_LIST(fcinfo);
  direction   = EXTPROTOCOL_VALIDATOR_GET_DIRECTION(fcinfo);
  /*
   * Dumb example 1: search each url for a substring
   * we don't want to be used in a url. in this example
   * it's 'secured_directory'.
   */
  for (i = 1 ; i <= nurls ; i++)
  {
    char *url = EXTPROTOCOL_VALIDATOR_GET_NTH_URL(fcinfo, i);

    if (strstr(url, "secured_directory") != 0)
    {
      ereport(ERROR,
       (errcode(ERRCODE_PROTOCOL_VIOLATION),
          errmsg("using 'secured_directory' in a url
            isn't allowed ")));
    }
  }
  /*
   * Dumb example 2: set a limit on the number of urls
   * used. In this example we limit readable external
   * tables that use our protocol to 2 urls max.
   */
  if(direction == EXT_VALIDATE_READ && nurls > 2)
  {
    ereport(ERROR,
      (errcode(ERRCODE_PROTOCOL_VIOLATION),
        errmsg("more than 2 urls aren't allowed in this protocol ")));
  }
  PG_RETURN_VOID();
}
/* --- utility functions --- */
static
DemoUri *ParseDemoUri(const char *uri_str)
{
  DemoUri *uri = (DemoUri *) palloc0(sizeof(DemoUri));
  int     protocol_len;

   uri->path = NULL;
   uri->protocol = NULL;
  /*
   * parse protocol
   */
  char *post_protocol = strstr(uri_str, "://");

  if(!post_protocol)
  {
    ereport(ERROR,
      (errcode(ERRCODE_SYNTAX_ERROR),
       errmsg("invalid protocol URI \'%s\'", uri_str),
       errOmitLocation(true)));
```

```
  }

  protocol_len = post_protocol - uri_str;
  uri->protocol = (char *)palloc0(protocol_len + 1);
  strncpy(uri->protocol, uri_str, protocol_len);

  /* make sure there is more to the uri string */
  if (strlen(uri_str) <= protocol_len)
    ereport(ERROR,
      (errcode(ERRCODE_SYNTAX_ERROR),
       errmsg("invalid myprot URI \'%s\' : missing path",
         uri_str),
      errOmitLocation(true)));

  /* parse path */
  uri->path = pstrdup(uri_str + protocol_len + strlen("://"));

  return uri;
}
static
void FreeDemoUri(DemoUri *uri)
{
  if (uri->path)
    pfree(uri->path);
  if (uri->protocol)
    pfree(uri->protocol);

  pfree(uri);
}
```

Managing Performance

The topics in this section cover Greenplum Database performance management, including how to monitor performance and how to configure workloads to prioritize resource utilization.

This section contains the following topics:

Defining Database Performance

Managing system performance includes measuring performance, identifying the causes of performance problems, and applying the tools and techniques available to you to remedy the problems.

Greenplum measures database performance based on the rate at which the database management system (DBMS) supplies information to requesters.

Understanding the Performance Factors

Several key performance factors influence database performance. Understanding these factors helps identify performance opportunities and avoid problems:

System Resources

Database performance relies heavily on disk I/O and memory usage. To accurately set performance expectations, you need to know the baseline performance of the hardware on which your DBMS is deployed. Performance of hardware components such as CPUs, hard disks, disk controllers, RAM, and network interfaces will significantly affect how fast your database performs.

Workload

The workload equals the total demand from the DBMS, and it varies over time. The total workload is a combination of user queries, applications, batch jobs, transactions, and system commands directed through the DBMS at any given time. For example, it can increase when month-end reports are run or decrease on weekends when most users are out of the office. Workload strongly influences database performance. Knowing your workload and peak demand times helps you plan for the most efficient use of your system resources and enables processing the largest possible workload.

Throughput

A system's throughput defines its overall capability to process data. DBMS throughput is measured in queries per second, transactions per second, or average response times. DBMS throughput is closely related to the processing capacity of the underlying systems (disk I/O, CPU speed, memory bandwidth, and so on), so it is important to know the throughput capacity of your hardware when setting DBMS throughput goals.

Contention

Contention is the condition in which two or more components of the workload attempt to use the system in a conflicting way — for example, multiple queries that try to update the same piece of data at the same time or multiple large workloads that compete for system resources. As contention increases, throughput decreases.

Optimization

DBMS optimizations can affect the overall system performance. SQL formulation, database configuration parameters, table design, data distribution, and so on enable the database query optimizer to create the most efficient access plans.

Determining Acceptable Performance

When approaching a performance tuning initiative, you should know your system's expected level of performance and define measurable performance requirements so you can accurately evaluate your system's performance. Consider the following when setting performance goals:

- *Baseline Hardware Performance*
- *Performance Benchmarks*

Baseline Hardware Performance

Most database performance problems are caused not by the database, but by the underlying systems on which the database runs. I/O bottlenecks, memory problems, and network issues can notably degrade database performance. Knowing the baseline capabilities of your hardware and operating system (OS) will help you identify and troubleshoot hardware-related problems before you explore database-level or query-level tuning initiatives.

See the *Greenplum Database Reference Guide* for information about running the `gpcheckperf` utility to validate hardware and network performance.

Performance Benchmarks

To maintain good performance or fix performance issues, you should know the capabilities of your DBMS on a defined workload. A benchmark is a predefined workload that produces a known result set. Periodically run the same benchmark tests to help identify system-related performance degradation over time. Use benchmarks to compare workloads and identify queries or applications that need optimization.

Many third-party organizations, such as the Transaction Processing Performance Council (TPC), provide benchmark tools for the database industry. TPC provides TPC-H, a decision support system that examines large volumes of data, executes queries with a high degree of complexity, and gives answers to critical business questions. For more information about TPC-H, go to:

http://www.tpc.org/tpch

Common Causes of Performance Issues

This section explains the troubleshooting processes for common performance issues and potential solutions to these issues.

Identifying Hardware and Segment Failures

The performance of Greenplum Database depends on the hardware and IT infrastructure on which it runs. Greenplum Database is comprised of several servers (hosts) acting together as one cohesive system (array); as a first step in diagnosing performance problems, ensure that all Greenplum Database segments are online. Greenplum Database's performance will be as fast as the slowest host in the array. Problems

with CPU utilization, memory management, I/O processing, or network load affect performance. Common hardware-related issues are:

- **Disk Failure** – Although a single disk failure should not dramatically affect database performance if you are using RAID, disk resynchronization does consume resources on the host with failed disks. The `gpcheckperf` utility can help identify segment hosts that have disk I/O issues.
- **Host Failure** – When a host is offline, the segments on that host are nonoperational. This means other hosts in the array must perform twice their usual workload because they are running the primary segments and multiple mirrors. If mirrors are not enabled, service is interrupted. Service is temporarily interrupted to recover failed segments. The `gpstate` utility helps identify failed segments.
- **Network Failure** – Failure of a network interface card, a switch, or DNS server can bring down segments. If host names or IP addresses cannot be resolved within your Greenplum array, these manifest themselves as interconnect errors in Greenplum Database. The `gpcheckperf` utility helps identify segment hosts that have network issues.
- **Disk Capacity** – Disk capacity on your segment hosts should never exceed 70 percent full. Greenplum Database needs some free space for runtime processing. To reclaim disk space that deleted rows occupy, run `VACUUM` after loads or updates. The *gp_toolkit* administrative schema has many views for checking the size of distributed database objects.

  See the *Greenplum Database Reference Guide* for information about checking database object sizes and disk space.

Managing Workload

A database system has a limited CPU capacity, memory, and disk I/O resources. When multiple workloads compete for access to these resources, database performance suffers. Resource management maximizes system throughput while meeting varied business requirements. Greenplum Database provides resource queues and resource groups to help you manage these system resources.

Resource queues and resource groups limit resource usage and the total number of concurrent queries executing in the particular queue or group. By assigning database roles to the appropriate queue or group, administrators can control concurrent user queries and prevent system overload. For more information about resource queues and resource groups, including selecting the appropriate scheme for your Greenplum Database environment, see *Managing Resources*.

Greenplum Database administrators should run maintenance workloads such as data loads and `VACUUM ANALYZE` operations after business hours. Do not compete with database users for system resources; perform administrative tasks at low-usage times.

Avoiding Contention

Contention arises when multiple users or workloads try to use the system in a conflicting way; for example, contention occurs when two transactions try to update a table simultaneously. A transaction that seeks a table-level or row-level lock will wait indefinitely for conflicting locks to be released. Applications should not hold transactions open for long periods of time, for example, while waiting for user input.

Maintaining Database Statistics

Greenplum Database uses a cost-based query optimizer that relies on database statistics. Accurate statistics allow the query optimizer to better estimate the number of rows retrieved by a query to choose the most efficient query plan. Without database statistics, the query optimizer cannot estimate how many records will be returned. The optimizer does not assume it has sufficient memory to perform certain operations such as aggregations, so it takes the most conservative action and does these operations by reading and writing from disk. This is significantly slower than doing them in memory. `ANALYZE` collects statistics about the database that the query optimizer needs.

> **Note:** When executing an SQL command with GPORCA, Greenplum Database issues a warning if the command performance could be improved by collecting statistics on a column or set of columns

referenced by the command. The warning is issued on the command line and information is added to the Greenplum Database log file. For information about collecting statistics on table columns, see the ANALYZE command in the *Greenplum Database Reference Guide*

Identifying Statistics Problems in Query Plans

Before you interpret a query plan for a query using EXPLAIN or `EXPLAIN ANALYZE`, familiarize yourself with the data to help identify possible statistics problems. Check the plan for the following indicators of inaccurate statistics:

- **Are the optimizer's estimates close to reality?** Run `EXPLAIN ANALYZE` and see if the number of rows the optimizer estimated is close to the number of rows the query operation returned .
- **Are selective predicates applied early in the plan?** The most selective filters should be applied early in the plan so fewer rows move up the plan tree.
- **Is the optimizer choosing the best join order?** When you have a query that joins multiple tables, make sure the optimizer chooses the most selective join order. Joins that eliminate the largest number of rows should be done earlier in the plan so fewer rows move up the plan tree.

See *Query Profiling* for more information about reading query plans.

Tuning Statistics Collection

The following configuration parameters control the amount of data sampled for statistics collection:

- `default_statistics_target`

These parameters control statistics sampling at the system level. It is better to sample only increased statistics for columns used most frequently in query predicates. You can adjust statistics for a particular column using the command:

`ALTER TABLE...SET STATISTICS`

For example:

```
ALTER TABLE sales ALTER COLUMN region SET STATISTICS 50;
```

This is equivalent to changing `default_statistics_target` for a particular column. Subsequent ANALYZE operations will then gather more statistics data for that column and produce better query plans as a result.

Optimizing Data Distribution

When you create a table in Greenplum Database, you must declare a distribution key that allows for even data distribution across all segments in the system. Because the segments work on a query in parallel, Greenplum Database will always be as fast as the slowest segment. If the data is unbalanced, the segments that have more data will return their results slower and therefore slow down the entire system.

Optimizing Your Database Design

Many performance issues can be improved by database design. Examine your database design and consider the following:

- Does the schema reflect the way the data is accessed?
- Can larger tables be broken down into partitions?
- Are you using the smallest data type possible to store column values?
- Are columns used to join tables of the same datatype?
- Are your indexes being used?

Greenplum Database Maximum Limits

To help optimize database design, review the maximum limits that Greenplum Database supports:

Table 59: Maximum Limits of Greenplum Database

Dimension	Limit
Database Size	Unlimited
Table Size	Unlimited, 128 TB per partition per segment
Row Size	1.6 TB (1600 columns * 1 GB)
Field Size	1 GB
Rows per Table	281474976710656 (2^48)
Columns per Table/View	1600
Indexes per Table	Unlimited
Columns per Index	32
Table-level Constraints per Table	Unlimited
Table Name Length	63 Bytes (Limited by *name* data type)

Dimensions listed as unlimited are not intrinsically limited by Greenplum Database. However, they are limited in practice to available disk space and memory/swap space. Performance may suffer when these values are unusually large.

Note:

There is a maximum limit on the number of objects (tables, indexes, and views, but not rows) that may exist at one time. This limit is 4294967296 (2^32).

Greenplum Database Memory Overview

Memory is a key resource for a Greenplum Database system and, when used efficiently, can ensure high performance and throughput. This topic describes how segment host memory is allocated between segments and the options available to administrators to configure memory.

A Greenplum Database segment host runs multiple PostgreSQL instances, all sharing the host's memory. The segments have an identical configuration and they consume similar amounts of memory, CPU, and disk IO simultaneously, while working on queries in parallel.

For best query throughput, the memory configuration should be managed carefully. There are memory configuration options at every level in Greenplum Database, from operating system parameters, to managing resources with resource queues and resource groups, to setting the amount of memory allocated to an individual query.

Segment Host Memory

On a Greenplum Database segment host, the available host memory is shared among all the processes executing on the computer, including the operating system, Greenplum Database segment instances, and other application processes. Administrators must determine what Greenplum Database and non-Greenplum Database processes share the hosts' memory and configure the system to use the memory efficiently. It is equally important to monitor memory usage regularly to detect any changes in the way host memory is consumed by Greenplum Database or other processes.

The following figure illustrates how memory is consumed on a Greenplum Database segment host when resource queue-based resource management is active.

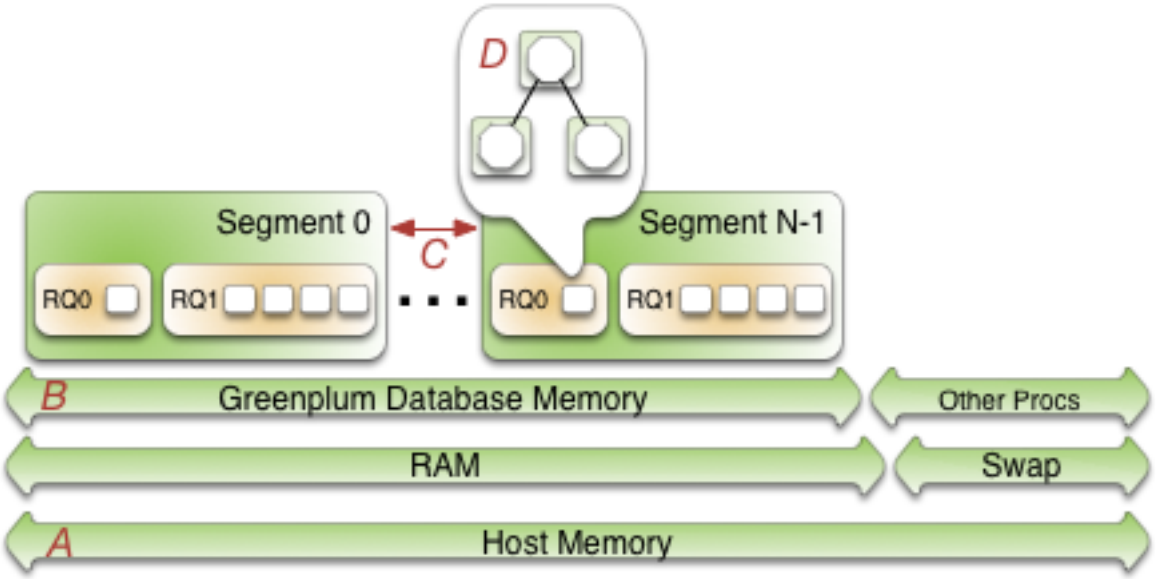


Figure 28: Greenplum Database Segment Host Memory

Beginning at the bottom of the illustration, the line labeled *A* represents the total host memory. The line directly above line *A* shows that the total host memory comprises both physical RAM and swap space.

The line labelled *B* shows that the total memory available must be shared by Greenplum Database and all other processes on the host. Non-Greenplum Database processes include the operating system and any other applications, for example system monitoring agents. Some applications may use a significant portion of memory and, as a result, you may have to adjust the number of segments per Greenplum Database host or the amount of memory per segment.

The segments (*C*) each get an equal share of the Greenplum Database Memory (*B*).

Within a segment, the currently active resource management scheme, *Resource Queues* or *Resource Groups*, governs how memory is allocated to execute a SQL statement. These constructs allow you to translate business requirements into execution policies in your Greenplum Database system and to guard against queries that could degrade performance. For an overview of resource groups and resource queues, refer to *Managing Resources*.

Options for Configuring Segment Host Memory

Host memory is the total memory shared by all applications on the segment host. You can configure the amount of host memory using any of the following methods:

- Add more RAM to the nodes to increase the physical memory.
- Allocate swap space to increase the size of virtual memory.
- Set the kernel parameters *vm.overcommit_memory* and *vm.overcommit_ratio* to configure how the operating system handles large memory allocation requests.

The physical RAM and OS configuration are usually managed by the platform team and system administrators. See the *Greenplum Database Installation Guide* for the recommended kernel parameters and for how to set the `/etc/sysctl.conf` file parameters.

The amount of memory to reserve for the operating system and other processes is workload dependent. The minimum recommendation for operating system memory is 32GB, but if there is much concurrency in Greenplum Database, increasing to 64GB of reserved memory may be required. The largest user of

operating system memory is SLAB, which increases as Greenplum Database concurrency and the number of sockets used increases.

The `vm.overcommit_memory` kernel parameter should always be set to 2, the only safe value for Greenplum Database.

The `vm.overcommit_ratio` kernel parameter sets the percentage of RAM that is used for application processes, the remainder reserved for the operating system. The default for Red Hat is 50 (50%). Setting this parameter too high may result in insufficient memory reserved for the operating system, which can cause segment host failure or database failure. Leaving the setting at the default of 50 is generally safe, but conservative. Setting the value too low reduces the amount of concurrency and the complexity of queries you can run at the same time by reducing the amount of memory available to Greenplum Database. When increasing `vm.overcommit_ratio`, it is important to remember to always reserve some memory for operating system activities.

Configuring vm.overcommit_ratio when Resource Group-Based Resource Management is Active

When resource group-based resource management is active, tune the operating system `vm.overcommit_ratio` as necessary. If your memory utilization is too low, increase the value; if your memory or swap usage is too high, decrease the setting.

Configuring vm.overcommit_ratio when Resource Queue-Based Resource Management is Active

To calculate a safe value for `vm.overcommit_ratio` when resource queue-based resource management is active, first determine the total memory available to Greenplum Database processes, `gp_vmem_rq`, with this formula:

```
gp_vmem_rq = ((SWAP + RAM) – (7.5GB + 0.05 * RAM)) / 1.7
```

where `SWAP` is the swap space on the host in GB, and `RAM` is the number of GB of RAM installed on the host. When resource queue-based resource management is active, use `gp_vmem_rq` to calculate the `vm.overcommit_ratio` value with this formula:

```
vm.overcommit_ratio = (RAM - 0.026 * gp_vmem_rq) / RAM
```

Configuring Greenplum Database Memory

Greenplum Database Memory is the amount of memory available to all Greenplum Database segment instances.

When you set up the Greenplum Database cluster, you determine the number of primary segments to run per host and the amount of memory to allocate for each segment. Depending on the CPU cores, amount of physical RAM, and workload characteristics, the number of segments is usually a value between 4 and 8. With segment mirroring enabled, it is important to allocate memory for the maximum number of primary segments executing on a host during a failure. For example, if you use the default grouping mirror configuration, a segment host failure doubles the number of acting primaries on the host that has the failed host's mirrors. Mirror configurations that spread each host's mirrors over multiple other hosts can lower the maximum, allowing more memory to be allocated for each segment. For example, if you use a block mirroring configuration with 4 hosts per block and 8 primary segments per host, a single host failure would cause other hosts in the block to have a maximum of 11 active primaries, compared to 16 for the default grouping mirror configuration.

Configuring Segment Memory when Resource Group-Based Resource Management is Active

When resource group-based resource management is active, the amount of memory allocated to each segment on a segment host is the memory available to Greenplum Database multiplied by the *gp_resource_group_memory_limit* server configuration parameter and divided by the number of active primary segments on the host. Use the following formula to calculate segment memory when using resource groups for resource management.

```
```

```
rg_perseg_mem = ((RAM * (vm.overcommit_ratio / 100) + SWAP) *
 gp_resource_group_memory_limit) / num_active_primary_segments
```

Resource groups expose additional configuration parameters that enable you to further control and refine the amount of memory allocated for queries.

Configuring Segment Memory when Resource Queue-Based Resource Management is Active

When resource queue-based resource management is active, the *gp_vmem_protect_limit* server configuration parameter value identifies the amount of memory to allocate to each segment. This value is estimated by calculating the memory available for all Greenplum Database processes and dividing by the maximum number of primary segments during a failure. If *gp_vmem_protect_limit* is set too high, queries can fail. Use the following formula to calculate a safe value for *gp_vmem_protect_limit*; provide the `gp_vmem_rq` value that you calculated earlier.

```
gp_vmem_protect_limit = gp_vmem_rq / max_acting_primary_segments
```

where `max_acting_primary_segments` is the maximum number of primary segments that could be running on a host when mirror segments are activated due to a host or segment failure.

gp_vmem_protect_limit does not affect segment memory when using resource groups for Greenplum Database resource management.

Resource queues expose additional configuration parameters that enable you to further control and refine the amount of memory allocated for queries.

Example Memory Configuration Calculations

This section provides example memory calculations for resource queues and resource groups for a Greenplum Database system with the following specifications:

- Total RAM = 256GB
- Swap = 64GB
- 8 primary segments and 8 mirror segments per host, in blocks of 4 hosts
- Maximum number of primaries per host during failure is 11

Resource Group Example

When resource group-based resource management is active in Greenplum Database, the usable memory available on a host is a function of the amount of RAM and swap space configured for the system, as well as the `vm.overcommit_ratio` system parameter setting:

```
total_node_usable_memory = RAM * (vm.overcommit_ratio / 100) + Swap
                         = 256GB * (50/100) + 64GB
                         = 192GB
```

Assuming the default `gp_resource_group_memory_limit` value (.7), the memory allocated to a Greenplum Database host with the example configuration is:

```
total_gp_memory = total_node_usable_memory * gp_resource_group_memory_limit
                = 192GB * .7
                = 134.4GB
```

The memory available to a Greenplum Database segment on a segment host is a function of the memory reserved for Greenplum on the host and the number of active primary segments on the host. On cluster startup:

```

```

```
gp_seg_memory = total_gp_memory / number_of_active_primary_segments
              = 134.4GB / 8
              = 16.8GB
```

Note that when 3 mirror segments switch to primary segments, the per-segment memory is still 16.8GB. Total memory usage on the segment host may approach:

```
total_gp_memory_with_primaries = 16.8GB * 11 = 184.8GB
```

Resource Queue Example

The `vm.overcommit_ratio` calculation for the example system when resource queue-based resource management is active in Greenplum Database follows:

```
gp_vmem_rq = ((SWAP + RAM) – (7.5GB + 0.05 * RAM)) / 1.7
        = ((64 + 256) - (7.5 + 0.05 * 256)) / 1.7
        = 176

vm.overcommit_ratio = (RAM - (0.026 * gp_vmem_rq)) / RAM
                    = (256 - (0.026 * 176)) / 256
                    = .982
```

You would set `vm.overcommit_ratio` of the example system to 98.

The `gp_vmem_protect_limit` calculation when resource queue-based resource management is active in Greenplum Database:

```
gp_vmem_protect_limit = gp_vmem_rq / maximum_acting_primary_segments
                      = 176 / 11
                      = 16GB
                      = 16384MB
```

You would set the `gp_vmem_protect_limit` server configuration parameter on the example system to 16384.

Managing Resources

Greenplum Database provides features to help you prioritize and allocate resources to queries according to business requirements and to prevent queries from starting when resources are unavailable.

You can use resource management features to limit the number of concurrent queries, the amount of memory used to execute a query, and the relative amount of CPU devoted to processing a query. Greenplum Database provides two schemes to manage resources - Resource Queues and Resource Groups.

> **Important:** Significant Greenplum Database performance degradation has been observed when enabling resource group-based workload management on RedHat 6.x and CentOS 6.x. This issue is caused by a Linux cgroup kernel bug. This kernel bug has been fixed in CentOS 7.x and Red Hat 7.x systems.
>
> If you use RedHat 6 and the performance with resource groups is acceptable for your use case, upgrade your kernel to version 2.6.32-696 or higher to benefit from other fixes to the cgroups implementation.

Either the resource queue or the resource group management scheme can be active in Greenplum Database; both schemes cannot be active at the same time.

Resource queues are enabled by default when you install your Greenplum Database cluster. While you can create and assign resource groups when resource queues are active, you must explicitly enable resource groups to start using that management scheme.

The following table summarizes some of the differences between Resource Queues and Resource Groups.

Metric	Resource Queues	Resource Groups
Concurrency	Managed at the query level	Managed at the transaction level
CPU	Specify query priority	Specify percentage of CPU resources; uses Linux Control Groups
Memory	Managed at the queue and operator level; users can over-subscribe	Managed at the transaction level, with enhanced allocation and tracking; users cannot over-subscribe
Memory Isolation	None	Memory is isolated between resource groups and between transactions within the same resource group
Users	Limits are applied only to non-admin users	Limits are applied to SUPERUSER and non-admin users alike
Queueing	Queue only when no slot available	Queue when no slot is available or not enough available memory
Query Failure	Query may fail immediately if not enough memory	Query may fail after reaching transaction fixed memory limit when no shared resource group memory exists and the transaction requests more memory
Limit Bypass	Limits are not enforced for SUPERUSER roles and certain operators and functions	Limits are not enforced on SET, RESET, and SHOW commands
External Components	None	Manage PL/Container CPU and memory resources

Using Resource Groups

You use resource groups to set and enforce CPU, memory, and concurrent transaction limits in Greenplum Database. After you define a resource group, you can then assign the group to one or more Greenplum Database roles, or to an external component such as PL/Container, in order to control the resources used by those roles or components.

When you assign a resource group to a role (a role-based resource group), the resource limits that you define for the group apply to all of the roles to which you assign the group. For example, the memory limit for a resource group identifies the maximum memory usage for all running transactions submitted by Greenplum Database users in all roles to which you assign the group.

Similarly, when you assign a resource group to an external component, the group limits apply to all running instances of the component. For example, if you create a resource group for a PL/Container external component, the memory limit that you define for the group specifies the maximum memory usage for all running instances of each PL/Container runtime to which you assign the group.

This topic includes the following subtopics:

- *Understanding Role and Component Resource Groups*
- *Resource Group Attributes and Limits*

Understanding Role and Component Resource Groups

Greenplum Database supports two types of resource groups: groups that manage resources for roles, and groups that manage resources for external components such as PL/Container.

The most common application for resource groups is to manage the number of active queries that different roles may execute concurrently in your Greenplum Database cluster. You can also manage the amount of CPU and memory resources that Greenplum allocates to each query.

Resource groups for roles use Linux control groups (cgroups) for CPU resource management. Greenplum Database tracks virtual memory internally for these resource groups using a memory auditor referred to as `vmtracker`.

When the user executes a query, Greenplum Database evaluates the query against a set of limits defined for the resource group. Greenplum Database executes the query immediately if the group's resource limits have not yet been reached and the query does not cause the group to exceed the concurrent transaction limit. If these conditions are not met, Greenplum Database queues the query. For example, if the maximum number of concurrent transactions for the resource group has already been reached, a subsequent query is queued and must wait until other queries complete before it runs. Greenplum Database may also execute a pending query when the resource group's concurrency and memory limits are altered to large enough values.

Within a resource group for roles, transactions are evaluated on a first in, first out basis. Greenplum Database periodically assesses the active workload of the system, reallocating resources and starting/queuing jobs as necessary.

You can also use resource groups to manage the CPU and memory resources of external components such as PL/Container. Resource groups for external components use Linux cgroups to manage both the total CPU and total memory resources for the component.

> **Note:** Containerized deployments of Greenplum Database, such as Greenplum for Kubernetes, might create a hierarchical set of nested cgroups to manage host system resources. The nesting of cgroups affects the Greenplum Database resource group limits for CPU percentage, CPU cores, and memory (except for Greenplum Database external components). The Greenplum Database resource group system resource limit is based on the quota for the parent group.
>
> For example, Greenplum Database is running in a cgroup demo, and the Greenplum Database cgroup is nested in the cgroup demo. If the cgroup demo is configured with a CPU limit of 60% of system CPU resources and the Greenplum Database resource group CPU limit is set 90%, the Greenplum Database limit of host system CPU resources is 54% (0.6 x 0.9).
>
> Nested cgroups do not affect memory limits for Greenplum Database external components such as PL/Container. Memory limits for external components can only be managed if the cgroup that is

used to manage Greenplum Database resources is not nested, the cgroup is configured as a top-level cgroup.

For information about configuring cgroups for use by resource groups, see *Configuring and Using Resource Groups*.

Resource Group Attributes and Limits

When you create a resource group, you:

- Specify the type of resource group by identifying how memory for the group is audited.
- Provide a set of limits that determine the amount of CPU and memory resources available to the group.

Resource group attributes and limits:

Limit Type	Description
MEMORY_AUDITOR	The memory auditor in use for the resource group. `vmtracker` (the default) is required if you want to assign the resource group to roles. Specify `cgroup` to assign the resource group to an external component.
CONCURRENCY	The maximum number of concurrent transactions, including active and idle transactions, that are permitted in the resource group.
CPU_RATE_LIMIT	The percentage of CPU resources available to this resource group.
CPUSET	The CPU cores to reserve for this resource group.
MEMORY_LIMIT	The percentage of reserved memory resources available to this resource group.
MEMORY_SHARED_QUOTA	The percentage of reserved memory to share across transactions submitted in this resource group.
MEMORY_SPILL_RATIO	The memory usage threshold for memory-intensive transactions. When a transaction reaches this threshold, it spills to disk.

Note: Resource limits are not enforced on `SET`, `RESET`, and `SHOW` commands.

Memory Auditor

The `MEMORY_AUDITOR` attribute specifies the type of resource group by identifying the memory auditor for the group. A resource group that specifies the `vmtracker` `MEMORY_AUDITOR` identifies a resource group for roles. A resource group specifying the `cgroup` `MEMORY_AUDITOR` identifies a resource group for external components.

The default `MEMORY_AUDITOR` is `vmtracker`.

The `MEMORY_AUDITOR` that you specify for a resource group determines if and how Greenplum Database uses the limit attributes to manage CPU and memory resources:

Limit Type	Resource Group for Roles	Resource Group for External Components
CONCURRENCY	Yes	No; must be zero (0)
CPU_RATE_LIMIT	Yes	Yes

Limit Type	Resource Group for Roles	Resource Group for External Components
CPUSET	Yes	Yes
MEMORY_LIMIT	Yes	Yes
MEMORY_SHARED_QUOTA	Yes	Component-specific
MEMORY_SPILL_RATIO	Yes	Component-specific

> **Note:** For queries managed by resource groups that are configured to use the `vmtracker` memory auditor, Greenplum Database supports the automatic termination of queries based on the amount of memory the queries are using. See the server configuration parameter *`runaway_detector_activation_percent`*.

Transaction Concurrency Limit

The `CONCURRENCY` limit controls the maximum number of concurrent transactions permitted for a resource group for roles.

> **Note:** The `CONCURRENCY` limit is not applicable to resource groups for external components and must be set to zero (0) for such groups.

Each resource group for roles is logically divided into a fixed number of slots equal to the `CONCURRENCY` limit. Greenplum Database allocates these slots an equal, fixed percentage of memory resources.

The default `CONCURRENCY` limit value for a resource group for roles is 20.

Greenplum Database queues any transactions submitted after the resource group reaches its `CONCURRENCY` limit. When a running transaction completes, Greenplum Database un-queues and executes the earliest queued transaction if sufficient memory resources exist.

You can set the server configuration parameter *`gp_resource_group_bypass`* to bypass a resource group concurrency limit.

You can set the server configuration parameter *`gp_resource_group_queuing_timeout`* to specify the amount of time a transaction remains in the queue before Greenplum Database cancels the transaction. The default timeout is zero, Greenplum queues transactions indefinitely.

CPU Limits

You configure the share of CPU resources to reserve for a resource group on a segment host by assigning specific CPU core(s) to the group, or by identifying the percentage of segment CPU resources to allocate to the group. Greenplum Database uses the `CPUSET` and `CPU_RATE_LIMIT` resource group limits to identify the CPU resource allocation mode. You must specify only one of these limits when you configure a resource group.

You may employ both modes of CPU resource allocation simultaneously in your Greenplum Database cluster. You may also change the CPU resource allocation mode for a resource group at runtime.

The *`gp_resource_group_cpu_limit`* server configuration parameter identifies the maximum percentage of system CPU resources to allocate to resource groups on each Greenplum Database segment host. This limit governs the maximum CPU usage of all resource groups on a segment host regardless of the CPU allocation mode configured for the group. The remaining unreserved CPU resources are used for the OS kernel and the Greenplum Database auxiliary daemon processes. The default `gp_resource_group_cpu_limit` value is .9 (90%).

> **Note:** The default `gp_resource_group_cpu_limit` value may not leave sufficient CPU resources if you are running other workloads on your Greenplum Database cluster nodes, so be sure to adjust this server configuration parameter accordingly.

> **Warning:** Avoid setting `gp_resource_group_cpu_limit` to a value higher than .9. Doing so may result in high workload queries taking near all CPU resources, potentially starving Greenplum Database auxiliary processes.

Assigning CPU Resources by Core

You identify the CPU cores that you want to reserve for a resource group with the `CPUSET` property. The CPU cores that you specify must be available in the system and cannot overlap with any CPU cores that you reserved for other resource groups. (Although Greenplum Database uses the cores that you assign to a resource group exclusively for that group, note that those CPU cores may also be used by non-Greenplum processes in the system.)

Specify a comma-separated list of single core numbers or number intervals when you configure `CPUSET`. You must enclose the core numbers/intervals in single quotes, for example, '1,3-4'.

When you assign CPU cores to `CPUSET` groups, consider the following:

- A resource group that you create with `CPUSET` uses the specified cores exclusively. If there are no running queries in the group, the reserved cores are idle and cannot be used by queries in other resource groups. Consider minimizing the number of `CPUSET` groups to avoid wasting system CPU resources.
- Consider keeping CPU core 0 unassigned. CPU core 0 is used as a fallback mechanism in the following cases:
  - `admin_group` and `default_group` require at least one CPU core. When all CPU cores are reserved, Greenplum Database assigns CPU core 0 to these default groups. In this situation, the resource group to which you assigned CPU core 0 shares the core with `admin_group` and `default_group`.
  - If you restart your Greenplum Database cluster with one node replacement and the node does not have enough cores to service all `CPUSET` resource groups, the groups are automatically assigned CPU core 0 to avoid system start failure.
- Use the lowest possible core numbers when you assign cores to resource groups. If you replace a Greenplum Database node and the new node has fewer CPU cores than the original, or if you back up the database and want to restore it on a cluster with nodes with fewer CPU cores, the operation may fail. For example, if your Greenplum Database cluster has 16 cores, assigning cores 1-7 is optimal. If you create a resource group and assign CPU core 9 to this group, database restore to an 8 core node will fail.

Resource groups that you configure with `CPUSET` have a higher priority on CPU resources. The maximum CPU resource usage percentage for all resource groups configured with `CPUSET` on a segment host is the number of CPU cores reserved divided by the number of all CPU cores, multiplied by 100.

When you configure `CPUSET` for a resource group, Greenplum Database disables `CPU_RATE_LIMIT` for the group and sets the value to -1.

> **Note:** You must configure `CPUSET` for a resource group *after* you have enabled resource group-based resource management for your Greenplum Database cluster.

Assigning CPU Resources by Percentage

The Greenplum Database node CPU percentage is divided equally among each segment on the Greenplum node. Each resource group that you configure with a `CPU_RATE_LIMIT` reserves the specified percentage of the segment CPU for resource management.

The minimum `CPU_RATE_LIMIT` percentage you can specify for a resource group is 1, the maximum is 100.

The sum of `CPU_RATE_LIMIT`s specified for all resource groups that you define in your Greenplum Database cluster must not exceed 100.

The maximum CPU resource usage for all resource groups configured with a `CPU_RATE_LIMIT` on a segment host is the minimum of:

- The number of non-reserved CPU cores divided by the number of all CPU cores, multiplied by 100, and
- The `gp_resource_group_cpu_limit` value.

CPU resource assignment for resource groups configured with a `CPU_RATE_LIMIT` is elastic in that Greenplum Database may allocate the CPU resources of an idle resource group to a busier one(s). In such situations, CPU resources are re-allocated to the previously idle resource group when that resource group next becomes active. If multiple resource groups are busy, they are allocated the CPU resources of any idle resource groups based on the ratio of their `CPU_RATE_LIMIT`s. For example, a resource group created with a `CPU_RATE_LIMIT` of 40 will be allocated twice as much extra CPU resource as a resource group that you create with a `CPU_RATE_LIMIT` of 20.

When you configure `CPU_RATE_LIMIT` for a resource group, Greenplum Database disables `CPUSET` for the group and sets the value to -1.

Memory Limits

When resource groups are enabled, memory usage is managed at the Greenplum Database node, segment, and resource group levels. You can also manage memory at the transaction level with a resource group for roles.

The *`gp_resource_group_memory_limit`* server configuration parameter identifies the maximum percentage of system memory resources to allocate to resource groups on each Greenplum Database segment host. The default `gp_resource_group_memory_limit` value is .7 (70%).

The memory resource available on a Greenplum Database node is further divided equally among each segment on the node. When resource group-based resource management is active, the amount of memory allocated to each segment on a segment host is the memory available to Greenplum Database multiplied by the `gp_resource_group_memory_limit` server configuration parameter and divided by the number of active primary segments on the host:

```
rg_perseg_mem = ((RAM * (vm.overcommit_ratio / 100) + SWAP) *
 gp_resource_group_memory_limit) / num_active_primary_segments
```

Each resource group may reserve a percentage of the segment memory for resource management. You identify this percentage via the `MEMORY_LIMIT` value that you specify when you create the resource group. The minimum `MEMORY_LIMIT` percentage you can specify for a resource group is 0, the maximum is 100. When `MEMORY_LIMIT` is 0, Greenplum Database reserves no memory for the resource group, but uses resource group global shared memory to fulfill all memory requests in the group. Refer to *Global Shared Memory* for more information about resource group global shared memory.

The sum of `MEMORY_LIMIT`s specified for all resource groups that you define in your Greenplum Database cluster must not exceed 100.

Additional Memory Limits for Role-based Resource Groups

If resource group memory is reserved for roles (non-zero `MEMORY_LIMIT`), the memory is further divided into fixed and shared components. The `MEMORY_SHARED_QUOTA` value that you specify when you create the resource group identifies the percentage of reserved resource group memory that may be shared among the currently running transactions. This memory is allotted on a first-come, first-served basis. A running transaction may use none, some, or all of the `MEMORY_SHARED_QUOTA`.

The minimum `MEMORY_SHARED_QUOTA` that you can specify is 0, the maximum is 100. The default `MEMORY_SHARED_QUOTA` is 80.

As mentioned previously, `CONCURRENCY` identifies the maximum number of concurrently running transactions permitted in a resource group for roles. If fixed memory is reserved by a resource group (non-zero `MEMORY_LIMIT`), it is divided into `CONCURRENCY` number of transaction slots. Each slot is allocated a

fixed, equal amount of the resource group memory. Greenplum Database guarantees this fixed memory to each transaction.

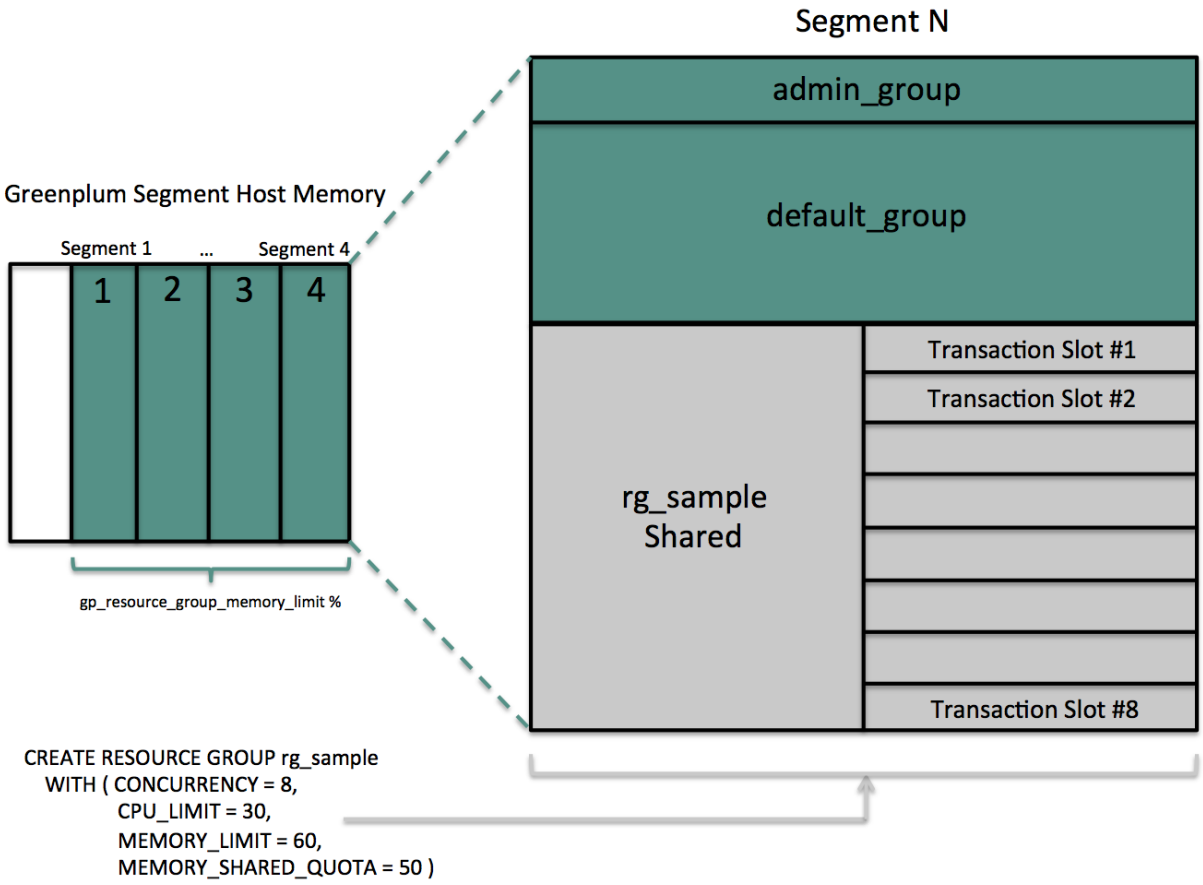


Figure 29: Resource Group Memory Allotments

When a query's memory usage exceeds the fixed per-transaction memory usage amount, Greenplum Database allocates available resource group shared memory to the query. The maximum amount of resource group memory available to a specific transaction slot is the sum of the transaction's fixed memory and the full resource group shared memory allotment.

Global Shared Memory

The sum of the `MEMORY_LIMIT`s configured for all resource groups (including the default `admin_group` and `default_group` groups) identifies the percentage of reserved resource group memory. If this sum is less than 100, Greenplum Database allocates any unreserved memory to a resource group global shared memory pool.

Resource group global shared memory is available only to resource groups that you configure with the `vmtracker` memory auditor.

When available, Greenplum Database allocates global shared memory to a transaction after first allocating slot and resource group shared memory (if applicable). Greenplum Database allocates resource group global shared memory to transactions on a first-come first-served basis.

> **Note:** Greenplum Database tracks, but does not actively monitor, transaction memory usage in resource groups. If the memory usage for a resource group exceeds its fixed memory allotment, a transaction in the resource group fails when *all* of these conditions are met:

- No available resource group shared memory exists.
- No available global shared memory exists.
- The transaction requests additional memory.

Greenplum Database uses resource group memory more efficiently when you leave some memory (for example, 10-20%) unallocated for the global shared memory pool. The availability of global shared memory also helps to mitigate the failure of memory-consuming or unpredicted queries.

Query Operator Memory

Most query operators are non-memory-intensive; that is, during processing, Greenplum Database can hold their data in allocated memory. When memory-intensive query operators such as join and sort process more data than can be held in memory, data is spilled to disk.

The *gp_resgroup_memory_policy* server configuration parameter governs the memory allocation and distribution algorithm for all query operators. Greenplum Database supports `eager-free` (the default) and `auto` memory policies for resource groups. When you specify the `auto` policy, Greenplum Database uses resource group memory limits to distribute memory across query operators, allocating a fixed size of memory to non-memory-intensive operators and the rest to memory-intensive operators. When the `eager_free` policy is in place, Greenplum Database distributes memory among operators more optimally by re-allocating memory released by operators that have completed their processing to operators in a later query stage.

`MEMORY_SPILL_RATIO` identifies the memory usage threshold for memory-intensive operators in a transaction. When this threshold is reached, a transaction spills to disk. Greenplum Database uses the `MEMORY_SPILL_RATIO` to determine the initial memory to allocate to a transaction.

You can specify an integer percentage value from 0 to 100 inclusive for `MEMORY_SPILL_RATIO`. The default `MEMORY_SPILL_RATIO` is 0.

When `MEMORY_SPILL_RATIO` is 0, Greenplum Database uses the *statement_mem* server configuration parameter value to control initial query operator memory.

> **Note:** When you set `MEMORY_LIMIT` to 0, `MEMORY_SPILL_RATIO` must also be set to 0.

You can selectively set the `MEMORY_SPILL_RATIO` on a per-query basis at the session level with the *memory_spill_ratio* server configuration parameter.

memory_spill_ratio and Low Memory Queries

A low `statement_mem` setting (for example, in the 10MB range) has been shown to increase the performance of queries with low memory requirements. Use the `memory_spill_ratio` and `statement_mem` server configuration parameters to override the setting on a per-query basis. For example:

```
SET memory_spill_ratio=0;
SET statement_mem='10 MB';
```

About Using Reserved Resource Group Memory vs. Using Resource Group Global Shared Memory

When you do not reserve memory for a resource group (`MEMORY_LIMIT` and `MEMORY_SPILL_RATIO` are set to 0):

- It increases the size of the resource group global shared memory pool.
- The resource group functions similarly to a resource queue, using the *statement_mem* server configuration parameter value to control initial query operator memory.
- Any query submitted in the resource group competes for resource group global shared memory on a first-come, first-served basis with queries running in other groups.
- There is no guarantee that Greenplum Database will be able to allocate memory for a query running in the resource group. The risk of a query in the group encountering an out of memory (OOM) condition increases when there are many concurrent queries consuming memory from the resource group global shared memory pool at the same time.

To reduce the risk of OOM for a query running in an important resource group, consider reserving some fixed memory for the group. While reserving fixed memory for a group reduces the size of the resource group global shared memory pool, this may be a fair tradeoff to reduce the risk of encountering an OOM condition in a query running in a critical resource group.

Other Memory Considerations

Resource groups for roles track all Greenplum Database memory allocated via the `palloc()` function. Memory that you allocate using the Linux `malloc()` function is not managed by these resource groups. To ensure that resource groups for roles are accurately tracking memory usage, avoid using `malloc()` to allocate large amounts of memory in custom Greenplum Database user-defined functions.

Using Greenplum Command Center to Manage Resource Groups

Using Pivotal Greenplum Command Center, an administrator can create and manage resource groups, change roles' resource groups, and create workload management rules.

Workload management assignment rules assign transactions to different resource groups based on user-defined criteria. If no assignment rule is matched, Greenplum Database assigns the transaction to the role's default resource group.

Refer to the *Greenplum Command Center documentation* for more information about creating and managing resource groups and workload management rules.

Configuring and Using Resource Groups

> **Important:** Significant Greenplum Database performance degradation has been observed when enabling resource group-based workload management on RedHat 6.x and CentOS 6.x systems. This issue is caused by a Linux cgroup kernel bug. This kernel bug has been fixed in CentOS 7.x and Red Hat 7.x systems.
>
> If you use RedHat 6 and the performance with resource groups is acceptable for your use case, upgrade your kernel to version 2.6.32-696 or higher to benefit from other fixes to the cgroups implementation.

Prerequisite

Greenplum Database resource groups use Linux Control Groups (cgroups) to manage CPU resources. Greenplum Database also uses cgroups to manage memory for resource groups for external components. With cgroups, Greenplum isolates the CPU and external component memory usage of your Greenplum processes from other processes on the node. This allows Greenplum to support CPU and external component memory usage restrictions on a per-resource-group basis.

For detailed information about cgroups, refer to the Control Groups documentation for your Linux distribution.

Complete the following tasks on each node in your Greenplum Database cluster to set up cgroups for use with resource groups:

1. If you are running the SuSE 11+ operating system on your Greenplum Database cluster nodes, you must enable swap accounting on each node and restart your Greenplum Database cluster. The `swapaccount` kernel boot parameter governs the swap accounting setting on SuSE 11+ systems. After setting this boot parameter, you must reboot your systems. For details, refer to the *Cgroup Swap Control* discussion in the SuSE 11 release notes. You must be the superuser or have `sudo` access to configure kernel boot parameters and reboot systems.
2. Create the Greenplum Database cgroups configuration file `/etc/cgconfig.d/gpdb.conf`. You must be the superuser or have `sudo` access to create this file:

   ```
   sudo vi /etc/cgconfig.d/gpdb.conf
   ```

3. Add the following configuration information to `/etc/cgconfig.d/gpdb.conf`:

   ```
   group gpdb {
        perm {
            task {
                uid = gpadmin;
                gid = gpadmin;
            }
            admin {
                uid = gpadmin;
                gid = gpadmin;
            }
        }
        cpu {
        }
        cpuacct {
        }
        cpuset {
        }
        memory {
        }
   }
   ```

   This content configures CPU, CPU accounting, CPU core set, and memory control groups managed by the `gpadmin` user. Greenplum Database uses the memory control group only for those resource groups created with the `cgroup` `MEMORY_AUDITOR`.
4. If not already installed and running, install the Control Groups operating system package and start the cgroups service on each Greenplum Database node. The commands that you run to perform these tasks will differ based on the operating system installed on the node. You must be the superuser or have `sudo` access to run these commands:

   - Redhat/CentOS 7.x systems:

     ```
     sudo yum install libcgroup-tools
     sudo cgconfigparser -l /etc/cgconfig.d/gpdb.conf
     ```

   - Redhat/CentOS 6.x systems:

     ```
     sudo yum install libcgroup
     sudo service cgconfig start
     ```

   - SuSE 11+ systems:

     ```
     sudo zypper install libcgroup-tools
     sudo cgconfigparser -l /etc/cgconfig.d/gpdb.conf
     ```

5. Identify the `cgroup` directory mount point for the node:

   ```
   grep cgroup /proc/mounts
   ```

The first line of output identifies the `cgroup` mount point.

6. Verify that you set up the Greenplum Database cgroups configuration correctly by running the following commands. Replace <cgroup_mount_point> with the mount point that you identified in the previous step:

```
ls -l <cgroup_mount_point>/cpu/gpdb
ls -l <cgroup_mount_point>/cpuacct/gpdb
ls -l <cgroup_mount_point>/cpuset/gpdb
ls -l <cgroup_mount_point>/memory/gpdb
```

   If these directories exist and are owned by `gpadmin:gpadmin`, you have successfully configured cgroups for Greenplum Database CPU resource management.

7. To automatically recreate Greenplum Database required cgroup hierarchies and parameters when your system is restarted, configure your system to enable the Linux cgroup service daemon `cgconfig.service` (Redhat/CentOS 7.x and SuSE 11+) or `cgconfig` (Redhat/CentOS 6.x) at node start-up. For example, configure one of the following cgroup service commands in your preferred service auto-start tool:

   - Redhat/CentOS 7.x and SuSE11+ systems:

     ```
     sudo systemctl enable cgconfig.service
     ```

     To start the service immediately (without having to reboot) enter:

     ```
     sudo systemctl start cgconfig.service
     ```

   - Redhat/CentOS 6.x systems:

     ```
     sudo chkconfig cgconfig on
     ```

   You may choose a different method to recreate the Greenplum Database resource group cgroup hierarchies.

Procedure

To use resource groups in your Greenplum Database cluster, you:

1. *Enable resource groups for your Greenplum Database cluster*.
2. *Create resource groups*.
3. *Assign the resource groups to one or more roles*.
4. *Use resource management system views to monitor and manage the resource groups*.

Enabling Resource Groups

When you install Greenplum Database, resource queues are enabled by default. To use resource groups instead of resource queues, you must set the *`gp_resource_manager`* server configuration parameter.

1. Set the `gp_resource_manager` server configuration parameter to the value `"group"`:

```
gpconfig -s gp_resource_manager
gpconfig -c gp_resource_manager -v "group"
```

2. Restart Greenplum Database:

```
gpstop
gpstart
```

Once enabled, any transaction submitted by a role is directed to the resource group assigned to the role, and is governed by that resource group's concurrency, memory, and CPU limits. Similarly, CPU and

memory usage by an external component is governed by the CPU and memory limits configured for the resource group assigned to the component.

Greenplum Database creates two default resource groups for roles named `admin_group` and `default_group`. When you enable resources groups, any role that was not explicitly assigned a resource group is assigned the default group for the role's capability. `SUPERUSER` roles are assigned the `admin_group`, non-admin roles are assigned the group named `default_group`.

The default resource groups `admin_group` and `default_group` are created with the following resource limits:

Limit Type	admin_group	default_group
CONCURRENCY	10	20
CPU_RATE_LIMIT	10	30
CPUSET	-1	-1
MEMORY_LIMIT	10	0
MEMORY_SHARED_QUOTA	80	80
MEMORY_SPILL_RATIO	0	0
MEMORY_AUDITOR	vmtracker	vmtracker

Keep in mind that the `CPU_RATE_LIMIT` and `MEMORY_LIMIT` values for the default resource groups `admin_group` and `default_group` contribute to the total percentages on a segment host. You may find that you need to adjust these limits for `admin_group` and/or `default_group` as you create and add new resource groups to your Greenplum Database deployment.

Creating Resource Groups

When you create a resource group for a role, you provide a name and a CPU resource allocation mode. You can optionally provide a concurrent transaction limit and memory limit, shared quota, and spill ratio values. Use the *`CREATE RESOURCE GROUP`* command to create a new resource group.

When you create a resource group for a role, you must provide a `CPU_RATE_LIMIT` or `CPUSET` limit value. These limits identify the percentage of Greenplum Database CPU resources to allocate to this resource group. You may specify a `MEMORY_LIMIT` to reserve a fixed amount of memory for the resource group. If you specify a `MEMORY_LIMIT` of 0, Greenplum Database uses global shared memory to fulfill all memory requirements for the resource group.

For example, to create a resource group named *rgroup1* with a CPU limit of 20, a memory limit of 25, and a memory spill ratio of 20:

```
=# CREATE RESOURCE GROUP rgroup1 WITH (CPU_RATE_LIMIT=20, MEMORY_LIMIT=25,
 MEMORY_SPILL_RATIO=20);
```

The CPU limit of 20 is shared by every role to which `rgroup1` is assigned. Similarly, the memory limit of 25 is shared by every role to which `rgroup1` is assigned. `rgroup1` utilizes the default `MEMORY_AUDITOR` `vmtracker` and the default `CONCURRENCY` setting of 20.

When you create a resource group for an external component, you must provide `CPU_RATE_LIMIT` or `CPUSET` and `MEMORY_LIMIT` limit values. You must also provide the `MEMORY_AUDITOR` and explicitly set `CONCURRENCY` to zero (0). For example, to create a resource group named *rgroup_extcomp* for which you reserve CPU core 1 and assign a memory limit of 15:

```
=# CREATE RESOURCE GROUP rgroup_extcomp WITH (MEMORY_AUDITOR=cgroup,
 CONCURRENCY=0,
     CPUSET='1', MEMORY_LIMIT=15);
```

The *ALTER RESOURCE GROUP* command updates the limits of a resource group. To change the limits of a resource group, specify the new values that you want for the group. For example:

```
=# ALTER RESOURCE GROUP rg_role_light SET CONCURRENCY 7;
=# ALTER RESOURCE GROUP exec SET MEMORY_SPILL_RATIO 25;
=# ALTER RESOURCE GROUP rgroup1 SET CPUSET '2,4';
```

Note: You cannot set or alter the `CONCURRENCY` value for the `admin_group` to zero (0).

The *DROP RESOURCE GROUP* command drops a resource group. To drop a resource group for a role, the group cannot be assigned to any role, nor can there be any transactions active or waiting in the resource group. Dropping a resource group for an external component in which there are running instances kills the running instances.

To drop a resource group:

```
=# DROP RESOURCE GROUP exec;
```

Configuring Automatic Query Termination Based on Memory Usage

When resource groups have a global shared memory pool, the server configuration parameter *runaway_detector_activation_percent* sets the percent of utilized global shared memory that triggers the termination of queries that are managed by resource groups that are configured to use the `vmtracker` memory auditor, such as `admin_group` and `default_group`.

Resource groups have a global shared memory pool when the sum of the `MEMORY_LIMIT` attribute values configured for all resource groups is less than 100. For example, if you have 3 resource groups configured with `MEMORY_LIMIT` values of 10 , 20, and 30, then global shared memory is 40% = 100% - (10% + 20% + 30%).

For information about global shared memory, see *Global Shared Memory*.

Assigning a Resource Group to a Role

When you create a resource group with the default `MEMORY_AUDITOR` `vmtracker`, the group is available for assignment to one or more roles (users). You assign a resource group to a database role using the `RESOURCE GROUP` clause of the *CREATE ROLE* or *ALTER ROLE* commands. If you do not specify a resource group for a role, the role is assigned the default group for the role's capability. `SUPERUSER` roles are assigned the `admin_group`, non-admin roles are assigned the group named `default_group`.

Use the `ALTER ROLE` or `CREATE ROLE` commands to assign a resource group to a role. For example:

```
=# ALTER ROLE bill RESOURCE GROUP rg_light;
=# CREATE ROLE mary RESOURCE GROUP exec;
```

You can assign a resource group to one or more roles. If you have defined a role hierarchy, assigning a resource group to a parent role does not propagate down to the members of that role group.

Note: You cannot assign a resource group that you create for an external component to a role.

If you wish to remove a resource group assignment from a role and assign the role the default group, change the role's group name assignment to `NONE`. For example:

```
=# ALTER ROLE mary RESOURCE GROUP NONE;
```

Monitoring Resource Group Status

Monitoring the status of your resource groups and queries may involve the following tasks:

- *Viewing Resource Group Limits*
- *Viewing Resource Group Query Status and CPU/Memory Usage*

- *Viewing the Resource Group Assigned to a Role*
- *Viewing a Resource Group's Running and Pending Queries*
- *Cancelling a Running or Queued Transaction in a Resource Group*

Viewing Resource Group Limits

The *gp_resgroup_config* gp_toolkit system view displays the current limits for a resource group. To view the limits of all resource groups:

```
=# SELECT * FROM gp_toolkit.gp_resgroup_config;
```

Viewing Resource Group Query Status and CPU/Memory Usage

The *gp_resgroup_status* gp_toolkit system view enables you to view the status and activity of a resource group. The view displays the number of running and queued transactions. It also displays the real-time CPU and memory usage of the resource group. To view this information:

```
=# SELECT * FROM gp_toolkit.gp_resgroup_status;
```

Viewing Resource Group CPU/Memory Usage Per Host

The *gp_resgroup_status_per_host* gp_toolkit system view enables you to view the real-time CPU and memory usage of a resource group on a per-host basis. To view this information:

```
=# SELECT * FROM gp_toolkit.gp_resgroup_status_per_host;
```

Viewing Resource Group CPU/Memory Usage Per Segment

The *gp_resgroup_status_per_segment* gp_toolkit system view enables you to view the real-time CPU and memory usage of a resource group on a per-segment, per-host basis. To view this information:

```
=# SELECT * FROM gp_toolkit.gp_resgroup_status_per_segment;
```

Viewing the Resource Group Assigned to a Role

To view the resource group-to-role assignments, perform the following query on the *pg_roles* and *pg_resgroup* system catalog tables:

```
=# SELECT rolname, rsgname FROM pg_roles, pg_resgroup
     WHERE pg_roles.rolresgroup=pg_resgroup.oid;
```

Viewing a Resource Group's Running and Pending Queries

To view a resource group's running queries, pending queries, and how long the pending queries have been queued, examine the *pg_stat_activity* system catalog table:

```
=# SELECT query, waiting, rsgname, rsgqueueduration
     FROM pg_stat_activity;
```

pg_stat_activity displays information about the user/role that initiated a query. A query that uses an external component such as PL/Container is composed of two parts: the query operator that runs in Greenplum Database and the UDF that runs in a PL/Container instance. Greenplum Database processes the query operators under the resource group assigned to the role that initiated the query. A UDF running in a PL/Container instance runs under the resource group assigned to the PL/Container runtime. The latter is not represented in the pg_stat_activity view; Greenplum Database does not have any insight into how external components such as PL/Container manage memory in running instances.

Cancelling a Running or Queued Transaction in a Resource Group

There may be cases when you want to cancel a running or queued transaction in a resource group. For example, you may want to remove a query that is waiting in the resource group queue but has not yet been executed. Or, you may want to stop a running query that is taking too long to execute, or one that is sitting idle in a transaction and taking up resource group transaction slots that are needed by other users.

By default, transactions can remain queued in a resource group indefinitely. If you want Greenplum Database to cancel a queued transaction after a specific amount of time, set the server configuration parameter *gp_resource_group_queuing_timeout*. When this parameter is set to a value (milliseconds) greater than 0, Greenplum cancels any queued transaction when it has waited longer than the configured timeout.

To manually cancel a running or queued transaction, you must first determine the process id (pid) associated with the transaction. Once you have obtained the process id, you can invoke `pg_cancel_backend()` to end that process, as shown below.

For example, to view the process information associated with all statements currently active or waiting in all resource groups, run the following query. If the query returns no results, then there are no running or queued transactions in any resource group.

```
=# SELECT rolname, g.rsgname, pid, waiting, state, query, datname
     FROM pg_roles, gp_toolkit.gp_resgroup_status g, pg_stat_activity
     WHERE pg_roles.rolresgroup=g.groupid
        AND pg_stat_activity.usename=pg_roles.rolname;
```

Sample partial query output:

```
 rolname | rsgname  | pid     | waiting | state  |          query
 | datname
---------+----------+---------+---------+--------+------------------------ -
+---------
  sammy  | rg_light |  31861  |    f    | idle   | SELECT * FROM mytesttbl;
 | testdb
  billy  | rg_light |  31905  |    t    | active | SELECT * FROM topten;
 | testdb
```

Use this output to identify the process id (`pid`) of the transaction you want to cancel, and then cancel the process. For example, to cancel the pending query identified in the sample output above:

```
=# SELECT pg_cancel_backend(31905);
```

You can provide an optional message in a second argument to `pg_cancel_backend()` to indicate to the user why the process was cancelled.

> **Note:**
>
> Do not use an operating system `KILL` command to cancel any Greenplum Database process.

Moving a Query to a Different Resource Group

A user with Greenplum Database superuser privileges can run the `gp_toolkit.pg_resgroup_move_query()` function to move a running query from one resource group to another, without stopping the query. Use this function to expedite a long-running query by moving it to a resource group with a higher resource allotment or availability.

> **Note:** You can move only an active or running query to a new resource group. You cannot move a queued or pending query that is in an idle state due to concurrency or memory limits.

`pg_resgroup_move_query()` requires the process id (pid) of the running query, as well as the name of the resource group to which you want to move the query. The signature of the function follows:

```
pg_resgroup_move_query( pid int4, group_name text );
```

You can obtain the pid of a running query from the `pg_stat_activity` system view as described in *Cancelling a Running or Queued Transaction in a Resource Group*. Use the `gp_toolkit.gp_resgroup_status` view to list the name, id, and status of each resource group.

When you invoke `pg_resgroup_move_query()`, the query is subject to the limits configured for the destination resource group:

- If the group has already reached its concurrent task limit, Greenplum Database queues the query until a slot opens.
- If the destination resource group does not have enough memory available to service the query's current memory requirements, Greenplum Database returns the error: `group <group_name> doesn't have enough memory ...`. In this situation, you may choose to increase the group shared memory allotted to the destination resource group, or perhaps wait a period of time for running queries to complete and then invoke the function again.

After Greenplum moves the query, there is no way to guarantee that a query currently running in the destination resource group does not exceed the group memory quota. In this situation, one or more running queries in the destination group may fail, including the moved query. Reserve enough resource group global shared memory to minimize the potential for this scenario to occur.

`pg_resgroup_move_query()` moves only the specified query to the destination resource group. Greenplum Database assigns subsequent queries that you submit in the session to the original resource group.

Note: Greenplum Database version 6.8 introduced support for moving a query to a different resource group.

- If you upgraded from a previous Greenplum 6.x installation, you must manually register the supporting functions for this feature, and grant access to the functions as follows:

```
CREATE FUNCTION gp_toolkit.pg_resgroup_check_move_query(IN session_id
 int, IN groupid oid, OUT session_mem int, OUT available_mem int)
RETURNS SETOF record
AS 'gp_resource_group', 'pg_resgroup_check_move_query'
VOLATILE LANGUAGE C;
GRANT EXECUTE ON FUNCTION
 gp_toolkit.pg_resgroup_check_move_query(int, oid, OUT int, OUT int)
 TO public;

CREATE FUNCTION gp_toolkit.pg_resgroup_move_query(session_id int4,
 groupid text)
RETURNS bool
AS 'gp_resource_group', 'pg_resgroup_move_query'
VOLATILE LANGUAGE C;
GRANT EXECUTE ON FUNCTION gp_toolkit.pg_resgroup_move_query(int4,
 text) TO public;
```

- If you register the supporting functions and then you downgrade your Greenplum Database installation to version 6.7.x or older, manually drop these functions as follows:

```
DROP FUNCTION gp_toolkit.pg_resgroup_check_move_query(IN int, IN oid,
 OUT int, OUT int);
DROP FUNCTION gp_toolkit.pg_resgroup_move_query(int4, text);
```

Resource Group Frequently Asked Questions

CPU

- **Why is CPU usage lower than the `CPU_RATE_LIMIT` configured for the resource group?**

  You may run into this situation when a low number of queries and slices are running in the resource group, and these processes are not utilizing all of the cores on the system.
- **Why is CPU usage for the resource group higher than the configured `CPU_RATE_LIMIT`?**

  This situation can occur in the following circumstances:

  - A resource group may utilize more CPU than its `CPU_RATE_LIMIT` when other resource groups are idle. In this situation, Greenplum Database allocates the CPU resource of an idle resource group to a busier one. This resource group feature is called CPU burst.
  - The operating system CPU scheduler may cause CPU usage to spike, then drop down. If you believe this might be occurring, calculate the average CPU usage within a given period of time (for example, 5 seconds) and use that average to determine if CPU usage is higher than the configured limit.

Memory

- **Why did my query return an "out of memory" error?**

  A transaction submitted in a resource group fails and exits when memory usage exceeds its fixed memory allotment, no available resource group shared memory exists, and the transaction requests more memory.
- **Why did my query return a "memory limit reached" error?**

  Greenplum Database automatically adjusts transaction and group memory to the new settings when you use `ALTER RESOURCE GROUP` to change a resource group's memory and/or concurrency limits. An "out of memory" error may occur if you recently altered resource group attributes and there is no longer a sufficient amount of memory available for a currently running query.
- **Why does the actual memory usage of my resource group exceed the amount configured for the group?**

  The actual memory usage of a resource group may exceed the configured amount when one or more queries running in the group is allocated memory from the global shared memory pool. (If no global shared memory is available, queries fail and do not impact the memory resources of other resource groups.)

  When global shared memory is available, memory usage may also exceed the configured amount when a transaction spills to disk. Greenplum Database statements continue to request memory when they start to spill to disk because:

  - Spilling to disk requires extra memory to work.
  - Other operators may continue to request memory.

  Memory usage grows in spill situations; when global shared memory is available, the resource group may eventually use up to 200-300% of its configured group memory limit.

Concurrency

- **Why is the number of running transactions lower than the `CONCURRENCY` limit configured for the resource group?**

  Greenplum Database considers memory availability before running a transaction, and will queue the transaction if there is not enough memory available to serve it. If you use `ALTER RESOURCE GROUP` to increase the `CONCURRENCY` limit for a resource group but do not also adjust memory limits, currently

running transactions may be consuming all allotted memory resources for the group. When in this state, Greenplum Database queues subsequent transactions in the resource group.

- **Why is the number of running transactions in the resource group higher than the configured `CONCURRENCY` limit?**

  The resource group may be running `SET` and `SHOW` commands, which bypass resource group transaction checks.

Using Resource Queues

Use Greenplum Database resource queues to prioritize and allocate resources to queries according to business requirements and to prevent queries from starting when resources are unavailable.

Resource queues are one tool to manage the degree of concurrency in a Greenplum Database system. Resource queues are database objects that you create with the `CREATE RESOURCE QUEUE` SQL statement. You can use them to manage the number of active queries that may execute concurrently, the amount of memory each type of query is allocated, and the relative priority of queries. Resource queues can also guard against queries that would consume too many resources and degrade overall system performance.

Each database role is associated with a single resource queue; multiple roles can share the same resource queue. Roles are assigned to resource queues using the `RESOURCE QUEUE` phrase of the `CREATE ROLE` or `ALTER ROLE` statements. If a resource queue is not specified, the role is associated with the default resource queue, `pg_default`.

When the user submits a query for execution, the query is evaluated against the resource queue's limits. If the query does not cause the queue to exceed its resource limits, then that query will run immediately. If the query causes the queue to exceed its limits (for example, if the maximum number of active statement slots are currently in use), then the query must wait until queue resources are free before it can run. Queries are evaluated on a first in, first out basis. If query prioritization is enabled, the active workload on the system is periodically assessed and processing resources are reallocated according to query priority (see *How Priorities Work*). Roles with the `SUPERUSER` attribute are exempt from resource queue limits. Superuser queries always run immediately regardless of limits imposed by their assigned resource queue.

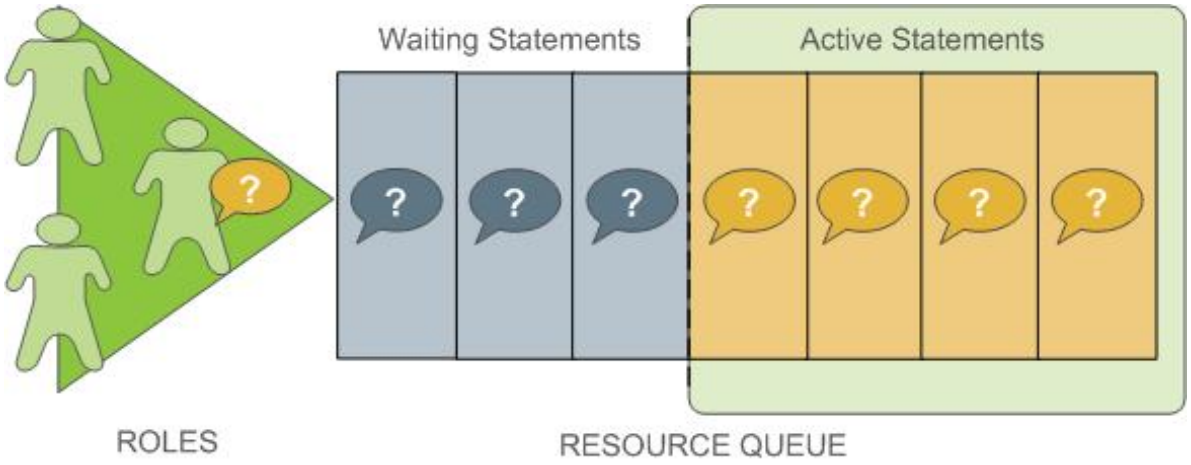


Figure 30: Resource Queue Process

Resource queues define classes of queries with similar resource requirements. Administratrors should create resource queues for the various types of workloads in their organization. For example, you could create resource queues for the following classes of queries, corresponding to different service level agreements:

- ETL queries
- Reporting queries
- Executive queries

A resource queue has the following characteristics:

MEMORY_LIMIT

The amount of memory used by all the queries in the queue (per segment). For example, setting *MEMORY_LIMIT* to 2GB on the ETL queue allows ETL queries to use up to 2GB of memory in each segment.

ACTIVE_STATEMENTS

The number of *slots* for a queue; the maximum concurrency level for a queue. When all slots are used, new queries must wait. Each query uses an equal amount of memory by default.

For example, the `pg_default` resource queue has *ACTIVE_STATEMENTS* = 20.

PRIORITY

The relative CPU usage for queries. This may be one of the following levels: `LOW`, `MEDIUM`, `HIGH`, `MAX`. The default level is `MEDIUM`. The query prioritization mechanism monitors the CPU usage of all the queries running in the system, and adjusts the CPU usage for each to conform to its priority level. For example, you could set `MAX` priority to the `executive` resource queue and `MEDIUM` to other queues to ensure that executive queries receive a greater share of CPU.

MAX_COST

Query plan cost limit.

The Greenplum Database optimizer assigns a numeric cost to each query. If the cost exceeds the `MAX_COST` value set for the resource queue, the query is rejected as too expensive.

Note: GPORCA and the Postgres Planner utilize different query costing models and may compute different costs for the same query. The Greenplum Database resource queue resource management scheme neither differentiates nor aligns costs between GPORCA and the Postgres Planner; it uses the literal cost value returned from the optimizer to throttle queries.

When resource queue-based resource management is active, use the `MEMORY_LIMIT` and `ACTIVE_STATEMENTS` limits for resource queues rather than configuring cost-based limits. Even when using GPORCA, Greenplum Database may fall back to using the Postgres Planner for certain queries, so using cost-based limits can lead to unexpected results.

The default configuration for a Greenplum Database system has a single default resource queue named `pg_default`. The `pg_default` resource queue has an `ACTIVE_STATEMENTS` setting of 20, no `MEMORY_LIMIT`, medium `PRIORITY`, and no set `MAX_COST`. This means that all queries are accepted and run immediately, at the same priority and with no memory limitations; however, only twenty queries may execute concurrently.

The number of concurrent queries a resource queue allows depends on whether the `MEMORY_LIMIT` parameter is set:

- If no `MEMORY_LIMIT` is set for a resource queue, the amount of memory allocated per query is the value of the *`statement_mem`* server configuration parameter. The maximum memory the resource queue can use is the product of `statement_mem` and `ACTIVE_STATEMENTS`.
- When a `MEMORY_LIMIT` is set on a resource queue, the number of queries that the queue can execute concurrently is limited by the queue's available memory.

A query admitted to the system is allocated an amount of memory and a query plan tree is generated for it. Each node of the tree is an operator, such as a sort or hash join. Each operator is a separate execution thread and is allocated a fraction of the overall statement memory, at minimum 100KB. If the plan has a large number of operators, the minimum memory required for operators can exceed the available memory and the query will be rejected with an insufficient memory error. Operators determine if they can complete

their tasks in the memory allocated, or if they must spill data to disk, in work files. The mechanism that allocates and controls the amount of memory used by each operator is called *memory quota*.

Not all SQL statements submitted through a resource queue are evaluated against the queue limits. By default only `SELECT`, `SELECT INTO`, `CREATE TABLE AS SELECT`, and `DECLARE CURSOR` statements are evaluated. If the server configuration parameter `resource_select_only` is set to *off*, then `INSERT`, `UPDATE`, and `DELETE` statements will be evaluated as well.

Also, an SQL statement that is run during the execution of an `EXPLAIN ANALYZE` command is excluded from resource queues.

Resource Queue Example

The default resource queue, `pg_default`, allows a maximum of 20 active queries and allocates the same amount of memory to each. This is generally not adequate resource control for production systems. To ensure that the system meets performance expectations, you can define classes of queries and assign them to resource queues configured to execute them with the concurrency, memory, and CPU resources best suited for that class of query.

The following illustration shows an example resource queue configuration for a Greenplum Database system with `gp_vmem_protect_limit` set to 8GB:

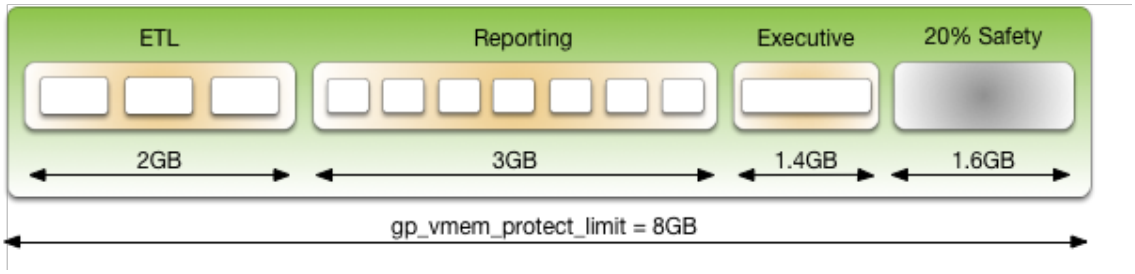


Figure 31: Resource Queue Configuration Example

This example has three classes of queries with different characteristics and service level agreements (SLAs). Three resource queues are configured for them. A portion of the segment memory is reserved as a safety margin.

Resource Queue Name	Active Statements	Memory Limit	Memory per Query
ETL	3	2GB	667MB
Reporting	7	3GB	429MB
Executive	1	1.4GB	1.4GB

The total memory allocated to the queues is 6.4GB, or 80% of the total segment memory defined by the `gp_vmem_protect_limit` server configuration parameter. Allowing a safety margin of 20% accommodates some operators and queries that are known to use more memory than they are allocated by the resource queue.

See the `CREATE RESOURCE QUEUE` and `CREATE/ALTER ROLE` statements in the *Greenplum Database Reference Guide* for help with command syntax and detailed reference information.

How Memory Limits Work

Setting `MEMORY_LIMIT` on a resource queue sets the maximum amount of memory that all active queries submitted through the queue can consume for a segment instance. The amount of memory allotted to a query is the queue memory limit divided by the active statement limit. (Use the memory limits in conjunction with statement-based queues rather than cost-based queues.) For example, if a queue has a memory limit of 2000MB and an active statement limit of 10, each query submitted through the queue

is allotted 200MB of memory by default. The default memory allotment can be overridden on a per-query basis using the `statement_mem` server configuration parameter (up to the queue memory limit). Once a query has started executing, it holds its allotted memory in the queue until it completes, even if during execution it actually consumes less than its allotted amount of memory.

You can use the `statement_mem` server configuration parameter to override memory limits set by the current resource queue. At the session level, you can increae `statement_mem` up to the resource queue's `MEMORY_LIMIT`. This will allow an individual query to use all of the memory allocated for the entire queue without affecting other resource queues.

The value of `statement_mem` is capped using the `max_statement_mem` configuration parameter (a superuser parameter). For a query in a resource queue with `MEMORY_LIMIT` set, the maximum value for `statement_mem` is `min(MEMORY_LIMIT, max_statement_mem)`. When a query is admitted, the memory allocated to it is subtracted from `MEMORY_LIMIT`. If `MEMORY_LIMIT` is exhausted, new queries in the same resource queue must wait. This happens even if `ACTIVE_STATEMENTS` has not yet been reached. Note that this can happen only when `statement_mem` is used to override the memory allocated by the resource queue.

For example, consider a resource queue named `adhoc` with the following settings:

- `MEMORY_LIMIT` is 1.5GB
- `ACTIVE_STATEMENTS` is 3

By default each statement submitted to the queue is allocated 500MB of memory. Now consider the following series of events:

1. User `ADHOC_1` submits query `Q1`, overriding`STATEMENT_MEM` to 800MB. The `Q1` statement is admitted into the system.
2. User `ADHOC_2` submits query `Q2`, using the default 500MB.
3. With `Q1` and `Q2` still running, user `ADHOC3` submits query `Q3`, using the default 500MB.

Queries `Q1` and `Q2` have used 1300MB of the queue's 1500MB. Therefore, `Q3` must wait for `Q1` or `Q2` to complete before it can run.

If `MEMORY_LIMIT` is not set on a queue, queries are admitted until all of the `ACTIVE_STATEMENTS` slots are in use, and each query can set an arbitrarily high `statement_mem`. This could lead to a resource queue using unbounded amounts of memory.

For more information on configuring memory limits on a resource queue, and other memory utilization controls, see *Creating Queues with Memory Limits*.

statement_mem and Low Memory Queries

A low `statement_mem` setting (for example, in the 1-3MB range) has been shown to increase the performance of queries with low memory requirements. Use the `statement_mem` server configuration parameter to override the setting on a per-query basis. For example:

```
SET statement_mem='2MB';
```

How Priorities Work

The `PRIORITY` setting for a resource queue differs from the `MEMORY_LIMIT` and `ACTIVE_STATEMENTS` settings, which determine whether a query will be admitted to the queue and eventually executed. The `PRIORITY` setting applies to queries after they become active. Active queries share available CPU resources as determined by the priority settings for its resource queue. When a statement from a high-priority queue enters the group of actively running statements, it may claim a greater share of the available CPU, reducing the share allocated to already-running statements in queues with a lesser priority setting.

The comparative size or complexity of the queries does not affect the allotment of CPU. If a simple, low-cost query is running simultaneously with a large, complex query, and their priority settings are the same,

they will be allocated the same share of available CPU resources. When a new query becomes active, the CPU shares will be recalculated, but queries of equal priority will still have equal amounts of CPU.

For example, an administrator creates three resource queues: *adhoc* for ongoing queries submitted by business analysts, *reporting* for scheduled reporting jobs, and *executive* for queries submitted by executive user roles. The administrator wants to ensure that scheduled reporting jobs are not heavily affected by unpredictable resource demands from ad-hoc analyst queries. Also, the administrator wants to make sure that queries submitted by executive roles are allotted a significant share of CPU. Accordingly, the resource queue priorities are set as shown:

- *adhoc* — Low priority
- *reporting* — High priority
- *executive* — Maximum priority

At runtime, the CPU share of active statements is determined by these priority settings. If queries 1 and 2 from the reporting queue are running simultaneously, they have equal shares of CPU. When an ad-hoc query becomes active, it claims a smaller share of CPU. The exact share used by the reporting queries is adjusted, but remains equal due to their equal priority setting:

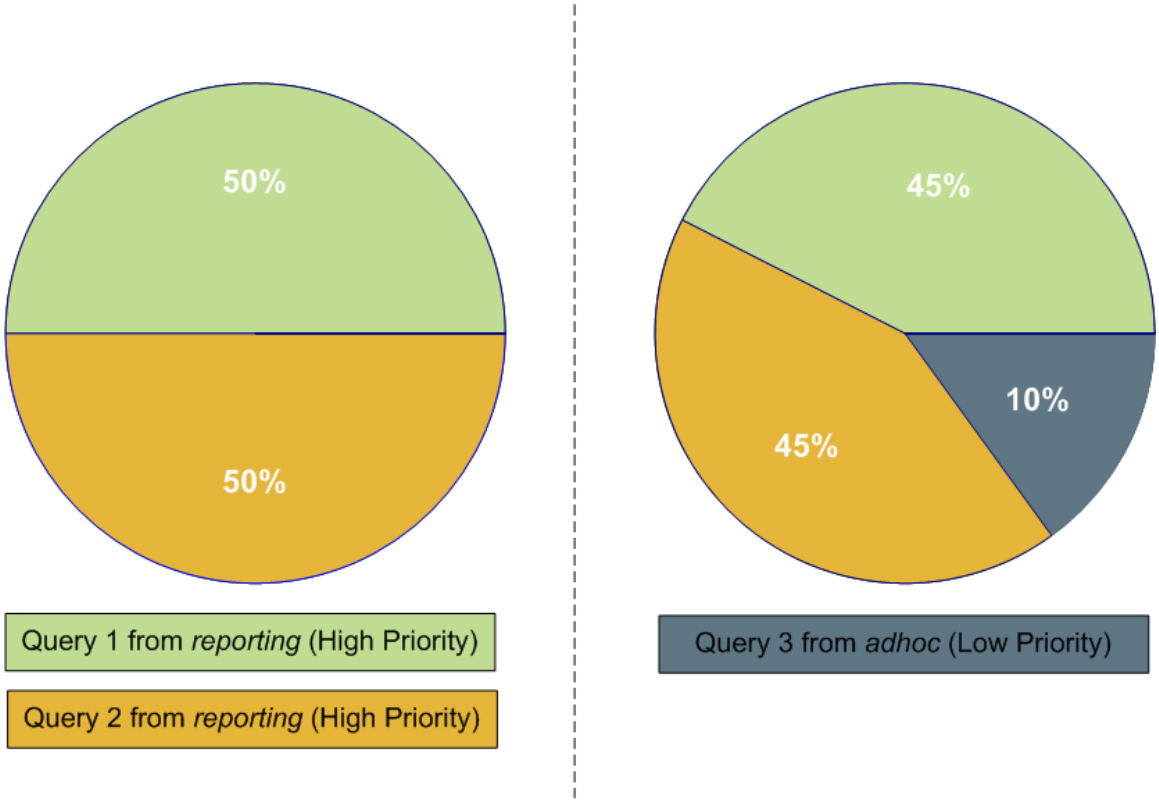


Figure 32: CPU share readjusted according to priority

Note:

The percentages shown in these illustrations are approximate. CPU usage between high, low and maximum priority queues is not always calculated in precisely these proportions.

When an executive query enters the group of running statements, CPU usage is adjusted to account for its maximum priority setting. It may be a simple query compared to the analyst and reporting queries, but until it is completed, it will claim the largest share of CPU.

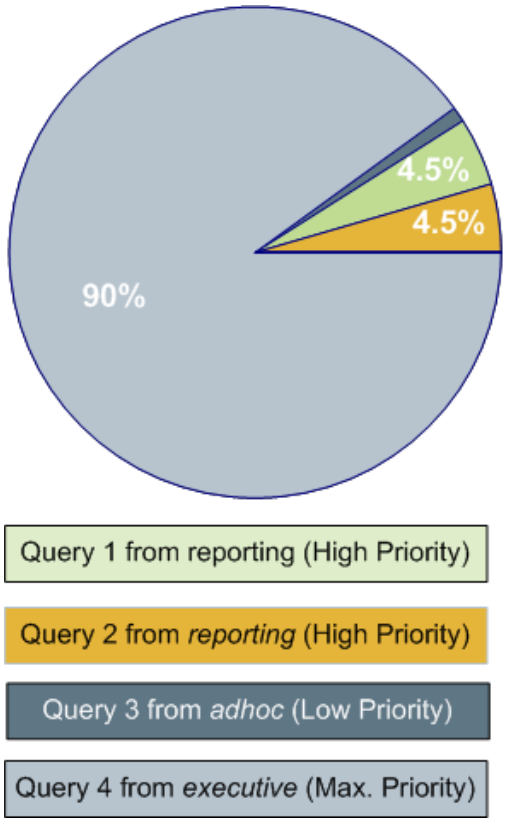


Figure 33: CPU share readjusted for maximum priority query

For more information about commands to set priorities, see *Setting Priority Levels*.

Steps to Enable Resource Management

Enabling and using resource management in Greenplum Database involves the following high-level tasks:

1. Configure resource management. See *Configuring Resource Management*.
2. Create the resource queues and set limits on them. See *Creating Resource Queues* and *Modifying Resource Queues*.
3. Assign a queue to one or more user roles. See *Assigning Roles (Users) to a Resource Queue*.
4. Use the resource management system views to monitor and manage the resource queues. See *Checking Resource Queue Status*.

Configuring Resource Management

Resource scheduling is enabled by default when you install Greenplum Database, and is required for all roles. The default resource queue, `pg_default`, has an active statement limit of 20, no memory limit, and a medium priority setting. Create resource queues for the various types of workloads.

To configure resource management

1. The following parameters are for the general configuration of resource queues:
   - `max_resource_queues` - Sets the maximum number of resource queues.
   - `max_resource_portals_per_transaction` - Sets the maximum number of simultaneously open cursors allowed per transaction. Note that an open cursor will hold an active query slot in a resource queue.

- `resource_select_only` - If set to *on*, then `SELECT`, `SELECT INTO`, `CREATE TABLE ASSELECT`, and `DECLARE CURSOR` commands are evaluated. If set to *off* `INSERT`, `UPDATE`, and `DELETE` commands will be evaluated as well.
- `resource_cleanup_gangs_on_wait` - Cleans up idle segment worker processes before taking a slot in the resource queue.
- `stats_queue_level` - Enables statistics collection on resource queue usage, which can then be viewed by querying the pg_stat_resqueues system view.

2. The following parameters are related to memory utilization:

- `gp_resqueue_memory_policy` - Enables Greenplum Database memory management features.

  In Greenplum Database 4.2 and later, the distribution algorithm `eager_free` takes advantage of the fact that not all operators execute at the same time. The query plan is divided into stages and Greenplum Database eagerly frees memory allocated to a previous stage at the end of that stage's execution, then allocates the eagerly freed memory to the new stage.

  When set to `none`, memory management is the same as in Greenplum Database releases prior to 4.1. When set to `auto`, query memory usage is controlled by `statement_mem` and resource queue memory limits.
- `statement_mem` and `max_statement_mem` - Used to allocate memory to a particular query at runtime (override the default allocation assigned by the resource queue). `max_statement_mem` is set by database superusers to prevent regular database users from over-allocation.
- `gp_vmem_protect_limit` - Sets the upper boundary that all query processes can consume and should not exceed the amount of physical memory of a segment host. When a segment host reaches this limit during query execution, the queries that cause the limit to be exceeded will be cancelled.
- `gp_vmem_idle_resource_timeout` and `gp_vmem_protect_segworker_cache_limit` - used to free memory on segment hosts held by idle database processes. Administrators may want to adjust these settings on systems with lots of concurrency.
- `shared_buffers` - Sets the amount of memory a Greenplum server instance uses for shared memory buffers. This setting must be at least 128 kilobytes and at least 16 kilobytes times `max_connections`. The value must not exceed the operating system shared memory maximum allocation request size, `shmmax` on Linux. See the *Greenplum Database Installation Guide* for recommended OS memory settings for your platform.

3. The following parameters are related to query prioritization. Note that the following parameters are all *local* parameters, meaning they must be set in the `postgresql.conf` files of the master and all segments:

- `gp_resqueue_priority` - The query prioritization feature is enabled by default.
- `gp_resqueue_priority_sweeper_interval` - Sets the interval at which CPU usage is recalculated for all active statements. The default value for this parameter should be sufficient for typical database operations.
- `gp_resqueue_priority_cpucores_per_segment` - Specifies the number of CPU cores allocated per segment instance. The default value is 4 for the master and segments.

  Each host checks its own `postgresql.conf` file for the value of this parameter. This parameter also affects the master node, where it should be set to a value reflecting the higher ratio of CPU cores. For example, on a cluster that has 10 CPU cores per host and 4 segments per host, you would specify these values for `gp_resqueue_priority_cpucores_per_segment`:

  10 for the master and standby master. Typically, only the master instance is on the master host.

  2.5 for segment instances on the segment hosts.

  If the parameter value is not set correctly, either the CPU might not be fully utilized, or query prioritization might not work as expected. For example, if the Greenplum Database cluster has fewer than one segment instance per CPU core on your segment hosts, make sure you adjust this value accordingly.

Actual CPU core utilization is based on the ability of Greenplum Database to parallelize a query and the resources required to execute the query.

Note: Any CPU core that is available to the operating system is included in the number of CPU cores. For example, virtual CPU cores are included in the number of CPU cores.

4. If you wish to view or change any of the resource management parameter values, you can use the `gpconfig` utility.
5. For example, to see the setting of a particular parameter:

```
$ gpconfig --show gp_vmem_protect_limit
```

6. For example, to set one value on all segment instances and a different value on the master:

```
$ gpconfig -c gp_resqueue_priority_cpucores_per_segment -v 2 -m 8
```

7. Restart Greenplum Database to make the configuration changes effective:

```
$ gpstop -r
```

Creating Resource Queues

Creating a resource queue involves giving it a name, setting an active query limit, and optionally a query priority on the resource queue. Use the `CREATE RESOURCE QUEUE` command to create new resource queues.

Creating Queues with an Active Query Limit

Resource queues with an `ACTIVE_STATEMENTS` setting limit the number of queries that can be executed by roles assigned to that queue. For example, to create a resource queue named *adhoc* with an active query limit of three:

```
=# CREATE RESOURCE QUEUE adhoc WITH (ACTIVE_STATEMENTS=3);
```

This means that for all roles assigned to the *adhoc* resource queue, only three active queries can be running on the system at any given time. If this queue has three queries running, and a fourth query is submitted by a role in that queue, that query must wait until a slot is free before it can run.

Creating Queues with Memory Limits

Resource queues with a `MEMORY_LIMIT` setting control the amount of memory for all the queries submitted through the queue. The total memory should not exceed the physical memory available per-segment. Set `MEMORY_LIMIT` to 90% of memory available on a per-segment basis. For example, if a host has 48 GB of physical memory and 6 segment instances, then the memory available per segment instance is 8 GB. You can calculate the recommended `MEMORY_LIMIT` for a single queue as 0.90*8=7.2 GB. If there are multiple queues created on the system, their total memory limits must also add up to 7.2 GB.

When used in conjunction with `ACTIVE_STATEMENTS`, the default amount of memory allotted per query is: `MEMORY_LIMIT / ACTIVE_STATEMENTS`. When used in conjunction with `MAX_COST`, the default amount of memory allotted per query is: `MEMORY_LIMIT * (query_cost / MAX_COST)`. Use `MEMORY_LIMIT` in conjunction with `ACTIVE_STATEMENTS` rather than with `MAX_COST`.

For example, to create a resource queue with an active query limit of 10 and a total memory limit of 2000MB (each query will be allocated 200MB of segment host memory at execution time):

```
=# CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20,
MEMORY_LIMIT='2000MB');
```

The default memory allotment can be overridden on a per-query basis using the `statement_mem` server configuration parameter, provided that `MEMORY_LIMIT` or `max_statement_mem` is not exceeded. For example, to allocate more memory to a particular query:

```
=> SET statement_mem='2GB';
=> SELECT * FROM my_big_table WHERE column='value' ORDER BY id;
=> RESET statement_mem;
```

As a general guideline, `MEMORY_LIMIT` for all of your resource queues should not exceed the amount of physical memory of a segment host. If workloads are staggered over multiple queues, it may be OK to oversubscribe memory allocations, keeping in mind that queries may be cancelled during execution if the segment host memory limit (`gp_vmem_protect_limit`) is exceeded.

Setting Priority Levels

To control a resource queue's consumption of available CPU resources, an administrator can assign an appropriate priority level. When high concurrency causes contention for CPU resources, queries and statements associated with a high-priority resource queue will claim a larger share of available CPU than lower priority queries and statements.

Priority settings are created or altered using the `WITH` parameter of the commands `CREATE RESOURCE QUEUE` and `ALTER RESOURCE QUEUE`. For example, to specify priority settings for the *adhoc* and *reporting* queues, an administrator would use the following commands:

```
=# ALTER RESOURCE QUEUE adhoc WITH (PRIORITY=LOW);
=# ALTER RESOURCE QUEUE reporting WITH (PRIORITY=HIGH);
```

To create the *executive* queue with maximum priority, an administrator would use the following command:

```
=# CREATE RESOURCE QUEUE executive WITH (ACTIVE_STATEMENTS=3, PRIORITY=MAX);
```

When the query prioritization feature is enabled, resource queues are given a `MEDIUM` priority by default if not explicitly assigned. For more information on how priority settings are evaluated at runtime, see *How Priorities Work*.

> **Important:** In order for resource queue priority levels to be enforced on the active query workload, you must enable the query prioritization feature by setting the associated server configuration parameters. See *Configuring Resource Management*.

Assigning Roles (Users) to a Resource Queue

Once a resource queue is created, you must assign roles (users) to their appropriate resource queue. If roles are not explicitly assigned to a resource queue, they will go to the default resource queue, `pg_default`. The default resource queue has an active statement limit of 20, no cost limit, and a medium priority setting.

Use the `ALTER ROLE` or `CREATE ROLE` commands to assign a role to a resource queue. For example:

```
=# ALTER ROLE name RESOURCE QUEUE queue_name;
=# CREATE ROLE name WITH LOGIN RESOURCE QUEUE queue_name;
```

A role can only be assigned to one resource queue at any given time, so you can use the `ALTER ROLE` command to initially assign or change a role's resource queue.

Resource queues must be assigned on a user-by-user basis. If you have a role hierarchy (for example, a group-level role) then assigning a resource queue to the group does not propagate down to the users in that group.

Superusers are always exempt from resource queue limits. Superuser queries will always run regardless of the limits set on their assigned queue.

Removing a Role from a Resource Queue

All users *must* be assigned to a resource queue. If not explicitly assigned to a particular queue, users will go into the default resource queue, `pg_default`. If you wish to remove a role from a resource queue and put them in the default queue, change the role's queue assignment to `none`. For example:

```
=# ALTER ROLE role_name RESOURCE QUEUE none;
```

Modifying Resource Queues

After a resource queue has been created, you can change or reset the queue limits using the `ALTER RESOURCE QUEUE` command. You can remove a resource queue using the `DROP RESOURCE QUEUE` command. To change the roles (users) assigned to a resource queue, *Assigning Roles (Users) to a Resource Queue*.

Altering a Resource Queue

The `ALTER RESOURCE QUEUE` command changes the limits of a resource queue. To change the limits of a resource queue, specify the new values you want for the queue. For example:

```
=# ALTER RESOURCE QUEUE adhoc WITH (ACTIVE_STATEMENTS=5);
=# ALTER RESOURCE QUEUE exec WITH (PRIORITY=MAX);
```

To reset active statements or memory limit to no limit, enter a value of `-1`. To reset the maximum query cost to no limit, enter a value of `-1.0`. For example:

```
=# ALTER RESOURCE QUEUE adhoc WITH (MAX_COST=-1.0, MEMORY_LIMIT='2GB');
```

You can use the `ALTER RESOURCE QUEUE` command to change the priority of queries associated with a resource queue. For example, to set a queue to the minimum priority level:

```
ALTER RESOURCE QUEUE webuser WITH (PRIORITY=MIN);
```

Dropping a Resource Queue

The `DROP RESOURCE QUEUE` command drops a resource queue. To drop a resource queue, the queue cannot have any roles assigned to it, nor can it have any statements waiting in the queue. See *Removing a Role from a Resource Queue* and *Clearing a Waiting Statement From a Resource Queue* for instructions on emptying a resource queue. To drop a resource queue:

```
=# DROP RESOURCE QUEUE name;
```

Checking Resource Queue Status

Checking resource queue status involves the following tasks:

- *Viewing Queued Statements and Resource Queue Status*
- *Viewing Resource Queue Statistics*
- *Viewing the Roles Assigned to a Resource Queue*
- *Viewing the Waiting Queries for a Resource Queue*
- *Clearing a Waiting Statement From a Resource Queue*
- *Viewing the Priority of Active Statements*
- *Resetting the Priority of an Active Statement*

Viewing Queued Statements and Resource Queue Status

The `gp_toolkit.gp_resqueue_status` view allows administrators to see status and activity for a resource queue. It shows how many queries are waiting to run and how many queries are currently active in the system from a particular resource queue. To see the resource queues created in the system, their limit attributes, and their current status:

```
=# SELECT * FROM gp_toolkit.gp_resqueue_status;
```

Viewing Resource Queue Statistics

If you want to track statistics and performance of resource queues over time, you can enable statistics collecting for resource queues. This is done by setting the following server configuration parameter in your master `postgresql.conf` file:

```
stats_queue_level = on
```

Once this is enabled, you can use the `pg_stat_resqueues` system view to see the statistics collected on resource queue usage. Note that enabling this feature does incur slight performance overhead, as each query submitted through a resource queue must be tracked. It may be useful to enable statistics collecting on resource queues for initial diagnostics and administrative planning, and then disable the feature for continued use.

See the Statistics Collector section in the PostgreSQL documentation for more information about collecting statistics in Greenplum Database.

Viewing the Roles Assigned to a Resource Queue

To see the roles assigned to a resource queue, perform the following query of the `pg_roles` and `gp_toolkit.gp_resqueue_status` system catalog tables:

```
=# SELECT rolname, rsqname FROM pg_roles,
          gp_toolkit.gp_resqueue_status
   WHERE pg_roles.rolresqueue=gp_toolkit.gp_resqueue_status.queueid;
```

You may want to create a view of this query to simplify future inquiries. For example:

```
=# CREATE VIEW role2queue AS
   SELECT rolname, rsqname FROM pg_roles, pg_resqueue
   WHERE pg_roles.rolresqueue=gp_toolkit.gp_resqueue_status.queueid;
```

Then you can just query the view:

```
=# SELECT * FROM role2queue;
```

Viewing the Waiting Queries for a Resource Queue

When a slot is in use for a resource queue, it is recorded in the `pg_locks` system catalog table. This is where you can see all of the currently active and waiting queries for all resource queues. To check that statements are being queued (even statements that are not waiting), you can also use the `gp_toolkit`.gp_locks_on_resqueue view. For example:

```
=# SELECT * FROM gp_toolkit.gp_locks_on_resqueue WHERE lorwaiting='true';
```

If this query returns no results, then that means there are currently no statements waiting in a resource queue.

Clearing a Waiting Statement From a Resource Queue

In some cases, you may want to clear a waiting statement from a resource queue. For example, you may want to remove a query that is waiting in the queue but has not been executed yet. You may also want to stop a query that has been started if it is taking too long to execute, or if it is sitting idle in a transaction and taking up resource queue slots that are needed by other users. To do this, you must first identify the statement you want to clear, determine its process id (pid), and then, use `pg_cancel_backend` with the process id to end that process, as shown below. An optional message to the process can be passed as the second parameter, to indicate to the user why the process was cancelled.

For example, to see process information about all statements currently active or waiting in all resource queues, run the following query:

```
=# SELECT rolname, rsqname, pg_locks.pid as pid, granted, state,
         query, datname
   FROM pg_roles, gp_toolkit.gp_resqueue_status, pg_locks,
        pg_stat_activity
   WHERE pg_roles.rolresqueue=pg_locks.objid
   AND pg_locks.objid=gp_toolkit.gp_resqueue_status.queueid
   AND pg_stat_activity.pid=pg_locks.pid
   AND pg_stat_activity.usename=pg_roles.rolname;
```

If this query returns no results, then that means there are currently no statements in a resource queue. A sample of a resource queue with two statements in it looks something like this:

```
rolname | rsqname |  pid  | granted | state  |         query          |
 datname
--------+---------+-------+---------+--------+------------------------
+---------
  sammy | webuser | 31861 | t       | idle   | SELECT * FROM testtbl; |
 namesdb
  daria | webuser | 31905 | f       | active | SELECT * FROM topten;  |
 namesdb
```

Use this output to identify the process id (pid) of the statement you want to clear from the resource queue. To clear the statement, you would then open a terminal window (as the `gpadmin` database superuser or as root) on the master host and cancel the corresponding process. For example:

```
=# pg_cancel_backend(31905)
```

Important: Do not use the operating system `KILL` command.

Viewing the Priority of Active Statements

The *gp_toolkit* administrative schema has a view called *gp_resq_priority_statement*, which lists all statements currently being executed and provides the priority, session ID, and other information.

This view is only available through the `gp_toolkit` administrative schema. See the *Greenplum Database Reference Guide* for more information.

Resetting the Priority of an Active Statement

Superusers can adjust the priority of a statement currently being executed using the built-in function `gp_adjust_priority(session_id, statement_count, priority)`. Using this function, superusers can raise or lower the priority of any query. For example:

```
=# SELECT gp_adjust_priority(752, 24905, 'HIGH')
```

To obtain the session ID and statement count parameters required by this function, superusers can use the `gp_toolkit` administrative schema view, *gp_resq_priority_statement*. From the view, use these values for the function parameters.

- The value of the `rqpsession` column for the `session_id` parameter
- The value of the `rqpcommand` column for the `statement_count` parameter
- The value of `rqppriority` column is the current priority. You can specify a string value of `MAX`, `HIGH`, `MEDIUM`, or `LOW` as the `priority`.

Note: The `gp_adjust_priority()` function affects only the specified statement. Subsequent statements in the same resource queue are executed using the queue's normally assigned priority.

Investigating a Performance Problem

This section provides guidelines for identifying and troubleshooting performance problems in a Greenplum Database system.

This topic lists steps you can take to help identify the cause of a performance problem. If the problem affects a particular workload or query, you can focus on tuning that particular workload. If the performance problem is system-wide, then hardware problems, system failures, or resource contention may be the cause.

Checking System State

Use the `gpstate` utility to identify failed segments. A Greenplum Database system will incur performance degradation when segment instances are down because other hosts must pick up the processing responsibilities of the down segments.

Failed segments can indicate a hardware failure, such as a failed disk drive or network card. Greenplum Database provides the hardware verification tool `gpcheckperf` to help identify the segment hosts with hardware issues.

Checking Database Activity

- *Checking for Active Sessions (Workload)*
- *Checking for Locks (Contention)*
- *Checking Query Status and System Utilization*

Checking for Active Sessions (Workload)

The *pg_stat_activity* system catalog view shows one row per server process; it shows the database OID, database name, process ID, user OID, user name, current query, time at which the current query began execution, time at which the process was started, client address, and port number. To obtain the most information about the current system workload, query this view as the database superuser. For example:

```
SELECT * FROM pg_stat_activity;
```

Note that the information does not update instantaneously.

Checking for Locks (Contention)

The *pg_locks* system catalog view allows you to see information about outstanding locks. If a transaction is holding a lock on an object, any other queries must wait for that lock to be released before they can continue. This may appear to the user as if a query is hanging.

Examine *pg_locks* for ungranted locks to help identify contention between database client sessions. *pg_locks* provides a global view of all locks in the database system, not only those relevant to the current database. You can join its relation column against `pg_class.oid` to identify locked relations (such as tables), but this works correctly only for relations in the current database. You can join the `pid` column to

the pg_stat_activity.pid to see more information about the session holding or waiting to hold a lock. For example:

```
SELECT locktype, database, c.relname, l.relation,
l.transactionid, l.pid, l.mode, l.granted,
a.query
        FROM pg_locks l, pg_class c, pg_stat_activity a
        WHERE l.relation=c.oid AND l.pid=a.pid
        ORDER BY c.relname;
```

If you use resource groups, queries that are waiting will also show in *pg_locks*. To see how many queries are waiting to run in a resource group, use the *gp_resgroup_status* system catalog view. For example:

```
SELECT * FROM gp_toolkit.gp_resgroup_status;
```

Similarly, if you use resource queues, queries that are waiting in a queue also show in *pg_locks*. To see how many queries are waiting to run from a resource queue, use the *gp_resqueue_status* system catalog view. For example:

```
SELECT * FROM gp_toolkit.gp_resqueue_status;
```

Checking Query Status and System Utilization

You can use system monitoring utilities such as ps, top, iostat, vmstat, netstat and so on to monitor database activity on the hosts in your Greenplum Database array. These tools can help identify Greenplum Database processes (postgres processes) currently running on the system and the most resource intensive tasks with regards to CPU, memory, disk I/O, or network activity. Look at these system statistics to identify queries that degrade database performance by overloading the system and consuming excessive resources. Greenplum Database's management tool gpssh allows you to run these system monitoring commands on several hosts simultaneously.

You can create and use the Greenplum Database *session_level_memory_consumption* view that provides information about the current memory utilization and idle time for sessions that are running queries on Greenplum Database. For information about the view, see *Viewing Session Memory Usage Information*.

You can enable a dedicated database, gpperfmon, in which data collection agents running on each segment host save query and system utilization metrics. Refer to the gpperfmon_install management utility reference in the *Greenplum Database Management Utility Reference Guide* for help creating the gpperfmon database and managing the agents. See documentation for the tables and views in the gpperfmon database in the *Greenplum Database Reference Guide*.

The optional Greenplum Command Center web-based user interface graphically displays query and system utilization metrics saved in the gpperfmon database. See the *Greenplum Command Center Documentation* web site for procedures to enable Greenplum Command Center.

Troubleshooting Problem Queries

If a query performs poorly, look at its query plan to help identify problems. The EXPLAIN command shows the query plan for a given query. See *Query Profiling* for more information about reading query plans and identifying problems.

When an out of memory event occurs during query execution, the Greenplum Database memory accounting framework reports detailed memory consumption of every query running at the time of the event. The information is written to the Greenplum Database segment logs.

Investigating Error Messages

Greenplum Database log messages are written to files in the pg_log directory within the master's or segment's data directory. Because the master log file contains the most information, you should always

check it first. Log files roll over daily and use the naming convention: `gpdb-*YYYY-MM-DD_hhmmss.csv*`. To locate the log files on the master host:

```
$ cd $MASTER_DATA_DIRECTORY/pg_log
```

Log lines have the format of:

```
timestamp | user | database | statement_id | con#cmd#
|:-LOG_LEVEL: log_message
```

You may want to focus your search for `WARNING`, `ERROR`, `FATAL` or `PANIC` log level messages. You can use the Greenplum utility `gplogfilter` to search through Greenplum Database log files. For example, when you run the following command on the master host, it checks for problem log messages in the standard logging locations:

```
$ gplogfilter -t
```

To search for related log entries in the segment log files, you can run `gplogfilter` on the segment hosts using `gpssh`. You can identify corresponding log entries by the *`statement_id`* or *`con#`* (session identifier). For example, to search for log messages in the segment log files containing the string `con6` and save output to a file:

```
gpssh -f seg_hosts_file -e 'source
/usr/local/greenplum-db/greenplum_path.sh ; gplogfilter -f
con6 /gpdata/*/pg_log/gpdb*.csv' > seglog.out
```

Gathering Information for Pivotal Customer Support

The Greenplum Magic Tool (GPMT) utility can run diagnostics and collect information from a Greenplum Database system. You can then send the information to Pivotal Customer Support to aid the diagnosis of Greenplum Database errors or system failures.

The GPMT utility is available from the *Pivotal Knowledge Base* on the *GPMT* page.

Chapter 4

Greenplum Database Security Configuration Guide

This guide describes how to secure a Greenplum Database system. The guide assumes knowledge of Linux/UNIX system administration and database management systems. Familiarity with structured query language (SQL) is helpful.

Because Greenplum Database is based on PostgreSQL 9.4, this guide assumes some familiarity with PostgreSQL. References to *PostgreSQL documentation* are provided throughout this guide for features that are similar to those in Greenplum Database.

This information is intended for system administrators responsible for administering a Greenplum Database system.

Securing the Database

Introduces Greenplum Database security topics.

The intent of security configuration is to configure the Greenplum Database server to eliminate as many security vulnerabilities as possible. This guide provides a baseline for minimum security requirements, and is supplemented by additional security documentation.

The essential security requirements fall into the following categories:

- *Authentication* covers the mechanisms that are supported and that can be used by the Greenplum database server to establish the identity of a client application.
- *Authorization* pertains to the privilege and permission models used by the database to authorize client access.
- *Auditing*, or log settings, covers the logging options available in Greenplum Database to track successful or failed user actions.
- *Data Encryption* addresses the encryption capabilities that are available for protecting data at rest and data in transit. This includes the security certifications that are relevant to the Greenplum Database.

Accessing a Kerberized Hadoop Cluster

You can use the Greenplum Platform Extension Framework (PXF) to read or write external tables referencing files in a Hadoop file system. If the Hadoop cluster is secured with Kerberos ("Kerberized"), you must configure Greenplum Database and PXF to allow users accessing external tables to authenticate with Kerberos. Refer to *Configuring PXF for Secure HDFS* for the procedure to perform this setup.

Platform Hardening

Platform hardening involves assessing and minimizing system vulnerability by following best practices and enforcing federal security standards. Hardening the product is based on the US Department of Defense (DoD) guidelines Security Template Implementation Guides (STIG). Hardening removes unnecessary packages, disables services that are not required, sets up restrictive file and directory permissions, removes unowned files and directories, performs authentication for single-user mode, and provides options for end users to configure the package to be compliant to the latest STIGs.

Greenplum Database Ports and Protocols

Lists network ports and protocols used within the Greenplum cluster.

Greenplum Database clients connect with TCP to the Greenplum master instance at the client connection port, 5432 by default. The listen port can be reconfigured in the `postgresql.conf` configuration file. Client connections use the PostgreSQL libpq API. The `psql` command-line interface, several Greenplum utilities, and language-specific programming APIs all either use the libpq library directly or implement the libpq protocol internally.

Each segment instance also has a client connection port, used solely by the master instance to coordinate database operations with the segments. The `gpstate -p` command, executed on the Greenplum master, lists the port assignments for the Greenplum master and the primary segments and mirrors. For example:

```
[gpadmin@mdw ~]$ gpstate -p
20190403:02:57:04:011030 gpstate:mdw:gpadmin-[INFO]:-Starting gpstate with
 args: -p
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-local
 Greenplum Version: 'postgres (Greenplum Database) 5.17.0 build
 commit:fc9a9d4cad8dd4037b9bc07bf837c0b958726103'
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-master Greenplum
 Version: 'PostgreSQL 8.3.23 (Greenplum Database 5.17.0 build
 commit:fc9a9d4cad8dd4037b9bc07bf837c0b958726103) on x86_64-pc-linux-gnu,
 compiled by GCC gcc (GCC) 6.2.0, 64-bit compiled on Feb 13 2019 15:26:34'
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-Obtaining Segment
 details from master...
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:--Master segment
 instance  /data/master/gpseg-1  port = 5432
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:--Segment instance port
 assignments
20190403:02:57:05:011030 gpstate:mdw:gpadmin-
[INFO]:----------------------------------
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   Host   Datadir
          Port
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw1   /data/
primary/gpseg0   20000
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw2   /data/mirror/
gpseg0    21000
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw1   /data/
primary/gpseg1   20001
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw2   /data/mirror/
gpseg1    21001
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw1   /data/
primary/gpseg2   20002
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw2   /data/mirror/
gpseg2    21002
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw2   /data/
primary/gpseg3   20000
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw3   /data/mirror/
gpseg3    21000
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw2   /data/
primary/gpseg4   20001
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw3   /data/mirror/
gpseg4    21001
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw2   /data/
primary/gpseg5   20002
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw3   /data/mirror/
gpseg5    21002
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw3   /data/
primary/gpseg6   20000
```

```
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw1   /data/mirror/
gpseg6    21000
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw3   /data/
primary/gpseg7   20001
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw1   /data/mirror/
gpseg7    21001
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw3   /data/
primary/gpseg8   20002
20190403:02:57:05:011030 gpstate:mdw:gpadmin-[INFO]:-   sdw1   /data/mirror/
gpseg8    21002
```

Additional Greenplum Database network connections are created for features such as standby replication, segment mirroring, statistics collection, and data exchange between segments. Some persistent connections are established when the database starts up and other transient connections are created during operations such as query execution. Transient connections for query execution processes, data movement, and statistics collection use available ports in the range 1025 to 65535 with both TCP and UDP protocols.

> **Note:** To avoid port conflicts between Greenplum Database and other applications when initializing Greenplum Database, do not specify Greenplum Database ports in the range specified by the operating system parameter `net.ipv4.ip_local_port_range`. For example, if `net.ipv4.ip_local_port_range = 10000 65535`, you could set the Greenplum Database base port numbers to values outside of that range:
>
> ```
> PORT_BASE = 6000
> MIRROR_PORT_BASE = 7000
> ```

Some add-on products and services that work with Greenplum Database have additional networking requirements. The following table lists ports and protocols used within the Greenplum cluster, and includes services and applications that integrate with Greenplum Database.

Table 60: Greenplum Database Ports and Protocols

Service	Protocol/Port	Description
Master SQL client connection	TCP 5432, libpq	SQL client connection port on the Greenplum master host. Supports clients using the PostgreSQL libpq API. Configurable.
Segment SQL client connection	varies, libpq	The SQL client connection port for a segment instance. Each primary and mirror segment on a host must have a unique port. Ports are assigned when the Greenplum system is initialized or expanded. The `gp_segment_configuration` system catalog records port numbers for each primary (p) or mirror (m) segment in the `port` column. Run `gpstate -p` to view the ports in use.
Segment mirroring port	varies, libpq	The port where a segment receives mirrored blocks from its primary. The port is assigned when the mirror is set up. The `gp_segment_configuration` system catalog records port numbers for each primary (p) or mirror (m) segment in the `port` column. Run `gpstate -p` to view the ports in use.
Greenplum Database Interconnect	UDP 1025-65535, dynamically allocated	The Interconnect transports database tuples between Greenplum segments during query execution.

Service	Protocol/Port	Description
Standby master client listener	TCP 5432, libpq	SQL client connection port on the standby master host. Usually the same as the master client connection port. Configure with the `gpinitstandby` utility `-P` option.
Standby master replicator	TCP 1025-65535, gpsyncmaster	The `gpsyncmaster` process on the master host establishes a connection to the secondary master host to replicate the master's log to the standby master.
Greenplum Database file load and transfer utilities: gpfdist, gpload.	TCP 8080, HTTP TCP 9000, HTTPS	The gpfdist file serving utility can run on Greenplum hosts or external hosts. Specify the connection port with the `-p` option when starting the server. The gpload utility runs one or more instances of gpfdist with ports or port ranges specified in a configuration file.
Gpperfmon agents	TCP 8888	Connection port for gpperfmon agents (`gpmmon` and `gpsmon`) executing on Greenplum Database hosts. Configure by setting the `gpperfmon_port` configuration variable in `postgresql.conf` on master and segment hosts.
Backup completion notification	TCP 25, TCP 587, SMTP	The `gpbackup` backup utility can optionally send email to a list of email addresses at completion of a backup. The SMTP service must be enabled on the Greenplum master host.
Greenplum Database secure shell (SSH): gpssh, gpscp, gpssh-exkeys, gppkg	TCP 22, SSH	Many Greenplum utilities use scp and ssh to transfer files between hosts and manage the Greenplum system within the cluster.
Greenplum Platform Extension Framework (PXF)	TCP 5888	The PXF Java service runs on port number 5888 on each Greenplum Database segment host.
Greenplum Command Center (GPCC)	TCP 28080, HTTP/ HTTPS, WebSocket (WS), Secure WebSocket (WSS)	The GPCC web server (`gpccws` process) executes on the Greenplum Database master host or standby master host. The port number is configured at installation time.
	TCP 8899, rcp port	A GPCC agent (`ccagent` process) on each Greenplum Database segment host connects to the GPCC rpc backend at port number 8899 on the GPCC web server host.
	UNIX domain socket, agent	Greenplum Database processes transmit datagrams to the GPCC agent (`ccagent` process) on each segment host using a UNIX domain socket.

Service	Protocol/Port	Description
GPText	TCP 2188 (base port)	ZooKeeper client ports. ZooKeeper uses a range of ports beginning at the base port number. The base port number and maximum port number are set in the GPText installation configuration file at installation time. The default base port number is 2188.
	TCP 18983 (base port)	GPText (Apache Solr) nodes. GPText nodes use a range of ports beginning at the base port number. The base port number and maximum port number are set in the GPText installation configuration file at installation time. The default base port number is 18983.
EMC Data Domain and DD Boost	TCP/UDP 111, NFS portmapper	Used to assign a random port for the mountd service used by NFS and DD Boost. The mountd service port can be statically assigned on the Data Domain server.
	TCP 2052	Main port used by NFS mountd. This port can be set on the Data Domain system using the `nfs set mountd-port` command .
	TCP 2049, NFS	Main port used by NFS. This port can be configured using the `nfs set server-port` command on the Data Domain server.
	TCP 2051, replication	Used when replication is configured on the Data Domain system. This port can be configured using the `replication modify` command on the Data Domain server.
Pgbouncer connection pooler	TCP, libpq	The pgbouncer connection pooler runs between libpq clients and Greenplum (or PostgreSQL) databases. It can be run on the Greenplum master host, but running it on a host outside of the Greenplum cluster is recommended. When it runs on a separate host, pgbouncer can act as a warm standby mechanism for the Greenplum master host, switching to the Greenplum standby host without requiring clients to reconfigure. Set the client connection port and the Greenplum master host address and port in the `pgbouncer.ini` configuration file.

Configuring Client Authentication

Describes the available methods for authenticating Greenplum Database clients.

When a Greenplum Database system is first initialized, the system contains one predefined superuser role. This role will have the same name as the operating system user who initialized the Greenplum Database system. This role is referred to as gpadmin. By default, the system is configured to only allow local connections to the database from the gpadmin role. If you want to allow any other roles to connect, or if you want to allow connections from remote hosts, you have to configure Greenplum Database to allow such connections. This section explains how to configure client connections and authentication to Greenplum Database.

Allowing Connections to Greenplum Database

Client access and authentication is controlled by a configuration file named `pg_hba.conf` (the standard PostgreSQL host-based authentication file). For detailed information about this file, see *The pg_hba.conf File* in the PostgreSQL documentation.

In Greenplum Database, the `pg_hba.conf` file of the master instance controls client access and authentication to your Greenplum system. The segments also have `pg_hba.conf` files, but these are already correctly configured to only allow client connections from the master host. The segments never accept outside client connections, so there is no need to alter the `pg_hba.conf` file on segments.

The general format of the `pg_hba.conf` file is a set of records, one per line. Blank lines are ignored, as is any text after a # comment character. A record is made up of a number of fields which are separated by spaces and/or tabs. Fields can contain white space if the field value is quoted. Records cannot be continued across lines.

A record can have one of seven formats:

```
local      database  user  auth-method  [auth-options]
host       database  user  address  auth-method  [auth-options]
hostssl    database  user  address  auth-method  [auth-options]
hostnossl  database  user  address  auth-method  [auth-options]
host       database  user  IP-address  IP-mask  auth-method  [auth-options]
hostssl    database  user  IP-address  IP-mask  auth-method  [auth-options]
hostnossl  database  user  IP-address  IP-mask  auth-method  [auth-options]
```

The meaning of the `pg_hba.conf` fields is as follows:

`local`

Matches connection attempts using UNIX-domain sockets. Without a record of this type, UNIX-domain socket connections are disallowed.

`host`

Matches connection attempts made using TCP/IP. Remote TCP/IP connections will not be possible unless the server is started with an appropriate value for the `listen_addresses` server configuration parameter. Greenplum Database by default allows connections from all hosts (`'*'`).

`hostssl`

Matches connection attempts made using TCP/IP, but only when the connection is made with SSL encryption. SSL must be enabled at server start time by setting the `ssl` configuration parameter to on. Requires SSL authentication be configured in `postgresql.conf`. See *Configuring postgresql.conf for SSL Authentication*.

`hostnossl`

Matches connection attempts made over TCP/IP that do not use SSL.

`database`

Specifies which database names this record matches. The value `all` specifies that it matches all databases. Multiple database names can be supplied by separating them with commas. A separate file containing database names can be specified by preceding the file name with `@`.

`user`

Specifies which database role names this record matches. The value `all` specifies that it matches all roles. If the specified role is a group and you want all members of that group to be included, precede the role name with a `+`. Multiple role names can be supplied by separating them with commas. A separate file containing role names can be specified by preceding the file name with `@`.

`address`

Specifies the client machine addresses that this record matches. This field can contain either a host name, an IP address range, or one of the special key words mentioned below.

An IP address range is specified using standard numeric notation for the range's starting address, then a slash (`/`) and a CIDR mask length. The mask length indicates the number of high-order bits of the client IP address that must match. Bits to the right of this should be zero in the given IP address. There must not be any white space between the IP address, the `/`, and the CIDR mask length.

Typical examples of an IPv4 address range specified this way are `172.20.143.89/32` for a single host, or `172.20.143.0/24` for a small network, or `10.6.0.0/16` for a larger one. An IPv6 address range might look like `::1/128` for a single host (in this case the IPv6 loopback address) or `fe80::7a31:c1ff:0000:0000/96` for a small network. `0.0.0.0/0` represents all IPv4 addresses, and `::0/0` represents all IPv6 addresses. To specify a single host, use a mask length of 32 for IPv4 or 128 for IPv6. In a network address, do not omit trailing zeroes.

An entry given in IPv4 format will match only IPv4 connections, and an entry given in IPv6 format will match only IPv6 connections, even if the represented address is in the IPv4-in-IPv6 range.

> **Note:** Entries in IPv6 format will be rejected if the host system C library does not have support for IPv6 addresses.

You can also write `all` to match any IP address, `samehost` to match any of the server's own IP addresses, or `samenet` to match any address in any subnet to which the server is directly connected.

If a host name is specified (an address that is not an IP address, IP range, or special key word is treated as a host name), that name is compared with the result of a reverse name resolution of the client IP address (for example, reverse DNS lookup, if DNS is used). Host name comparisons are case insensitive. If there is a match, then a forward name resolution (for example, forward DNS lookup) is performed on the host name to check whether any of the addresses it resolves to are equal to the client IP address. If both directions match, then the entry is considered to match.

The host name that is used in `pg_hba.conf` should be the one that address-to-name resolution of the client's IP address returns, otherwise the line won't be matched. Some

host name databases allow associating an IP address with multiple host names, but the operating system will only return one host name when asked to resolve an IP address.

A host name specification that starts with a dot (.) matches a suffix of the actual host name. So `.example.com` would match `foo.example.com` (but not just `example.com`).

When host names are specified in `pg_hba.conf`, you should ensure that name resolution is reasonably fast. It can be advantageous to set up a local name resolution cache such as `nscd`. Also, you can enable the server configuration parameter `log_hostname` to see the client host name instead of the IP address in the log.

IP-address
IP-mask

These two fields can be used as an alternative to the CIDR address notation. Instead of specifying the mask length, the actual mask is specified in a separate column. For example, `255.0.0.0` represents an IPv4 CIDR mask length of 8, and `255.255.255.255` represents a CIDR mask length of 32.

auth-method

Specifies the authentication method to use when a connection matches this record. See *Authentication Methods* for options.

auth-options

After the *`auth-method`* field, there can be field(s) of the form *`name=value`* that specify options for the authentication method. Details about which options are available for which authentication methods are described in *Authentication Methods*.

Files included by @ constructs are read as lists of names, which can be separated by either whitespace or commas. Comments are introduced by #, just as in `pg_hba.conf`, and nested @ constructs are allowed. Unless the file name following @ is an absolute path, it is taken to be relative to the directory containing the referencing file.

The `pg_hba.conf` records are examined sequentially for each connection attempt, so the order of the records is significant. Typically, earlier records will have tight connection match parameters and weaker authentication methods, while later records will have looser match parameters and stronger authentication methods. For example, you might wish to use `trust` authentication for local TCP/IP connections but require a password for remote TCP/IP connections. In this case a record specifying `trust` authentication for connections from 127.0.0.1 would appear before a record specifying `password` authentication for a wider range of allowed client IP addresses.

The `pg_hba.conf` file is read on start-up and when the main server process receives a SIGHUP signal. If you edit the file on an active system, you must reload the file using this command:

```
$ gpstop -u
```

Caution: For a more secure system, remove records for remote connections that use `trust` authentication from the `pg_hba.conf` file. `trust` authentication grants any user who can connect to the server access to the database using any role they specify. You can safely replace `trust` authentication with `ident` authentication for local UNIX-socket connections. You can also use `ident` authentication for local and remote TCP clients, but the client host must be running an ident service and you must `trust` the integrity of that machine.

Editing the pg_hba.conf File

Initially, the `pg_hba.conf` file is set up with generous permissions for the gpadmin user and no database access for other Greenplum Database roles. You will need to edit the `pg_hba.conf` file to enable users' access to databases and to secure the gpadmin user. Consider removing entries that have `trust` authentication, since they allow anyone with access to the server to connect with any role they choose. For local (UNIX socket) connections, use `ident` authentication, which requires the operating system user to match the role specified. For local and remote TCP connections, `ident` authentication requires the client's

host to run an indent service. You could install an ident service on the master host and then use `ident` authentication for local TCP connections, for example `127.0.0.1/28`. Using `ident` authentication for remote TCP connections is less secure because it requires you to trust the integrity of the ident service on the client's host.

> **Note:** Greenplum Command Center provides an interface for editing the `pg_hba.conf` file. It verifies entries before you save them, keeps a version history of the file so that you can reload a previous version of the file, and reloads the file into Greenplum Database.

This example shows how to edit the `pg_hba.conf` file on the master host to allow remote client access to all databases from all roles using encrypted password authentication.

To edit `pg_hba.conf`:

1. Open the file `$MASTER_DATA_DIRECTORY/pg_hba.conf` in a text editor.
2. Add a line to the file for each type of connection you want to allow. Records are read sequentially, so the order of the records is significant. Typically, earlier records will have tight connection match parameters and weaker authentication methods, while later records will have looser match parameters and stronger authentication methods. For example:

```
# allow the gpadmin user local access to all databases
# using ident authentication
local   all   gpadmin   ident         sameuser
host    all   gpadmin   127.0.0.1/32  ident
host    all   gpadmin   ::1/128       ident
# allow the 'dba' role access to any database from any
# host with IP address 192.168.x.x and use md5 encrypted
# passwords to authenticate the user
# Note that to use SHA-256 encryption, replace md5 with
# password in the line below
host    all   dba   192.168.0.0/32  md5
```

Authentication Methods

- *Basic Authentication*
- *GSSAPI Authentication*
- *LDAP Authentication*
- *SSL Client Authentication*
- *PAM-Based Authentication*
- *Radius Authentication*

Basic Authentication

`Reject`

Reject the connections with the matching parameters. You should typically use this to restrict access from specific hosts or insecure connections.

`Ident`

Authenticates based on the client's operating system user name. This is secure for local socket connections. Using `ident` for TCP connections from remote hosts requires that the client's host is running an ident service. The `ident` authentication method should only be used with remote hosts on a trusted, closed network.

`md5`

Require the client to supply a double-MD5-hashed password for authentication.

`password`

Require the client to supply an unencrypted password for authentication. Since the password is sent in clear text over the network, this should not be used on untrusted networks.

The password-based authentication methods are `md5` and `password`. These methods operate similarly except for the way that the password is sent across the connection: MD5-hashed and clear-text respectively.

If you are at all concerned about password "sniffing" attacks then `md5` is preferred. Plain `password` should always be avoided if possible. If the connection is protected by SSL encryption then `password` can be used safely (although SSL certificate authentication might be a better choice if you are depending on using SSL).

Following are some sample `pg_hba.conf` basic authentication entries:

```
hostnossl    all   all        0.0.0.0   reject
hostssl      all   testuser   0.0.0.0/0 md5
local        all   gpuser                ident
```

GSSAPI Authentication

GSSAPI is an industry-standard protocol for secure authentication defined in RFC 2743. Greenplum Database supports GSSAPI with Kerberos authentication according to RFC 1964. GSSAPI provides automatic authentication (single sign-on) for systems that support it. The authentication itself is secure, but the data sent over the database connection will be sent unencrypted unless SSL is used.

The `gss` authentication method is only available for TCP/IP connections.

When GSSAPI uses Kerberos, it uses a standard principal in the format *`servicename/hostname@realm`*. The Greenplum Database server will accept any principal that is included in the keytab file used by the server, but care needs to be taken to specify the correct principal details when making the connection from the client using the `krbsrvname` connection parameter. (See *Connection Parameter Key Words* in the PostgreSQL documentation.) In most environments, this parameter never needs to be changed. Some Kerberos implementations might require a different service name, such as Microsoft Active Directory, which requires the service name to be in upper case (POSTGRES).

`hostname` is the fully qualified host name of the server machine. The service principal's realm is the preferred realm of the server machine.

Client principals must have their Greenplum Database user name as their first component, for example `gpusername@realm`. Alternatively, you can use a user name mapping to map from the first component of the principal name to the database user name. By default, Greenplum Database does not check the realm of the client. If you have cross-realm authentication enabled and need to verify the realm, use the `krb_realm` parameter, or enable `include_realm` and use user name mapping to check the realm.

Make sure that your server keytab file is readable (and preferably only readable) by the `gpadmin` server account. The location of the key file is specified by the *krb_server_keyfile* configuration parameter. For security reasons, it is recommended to use a separate keytab just for the Greenplum Database server rather than opening up permissions on the system keytab file.

The keytab file is generated by the Kerberos software; see the Kerberos documentation for details. The following example is for MIT-compatible Kerberos 5 implementations:

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

When connecting to the database make sure you have a ticket for a principal matching the requested database user name. For example, for database user name `fred`, principal `fred@EXAMPLE.COM` would be able to connect. To also allow principal `fred/users.example.com@EXAMPLE.COM`, use a user name map, as described in *User Name Maps* in the PostgreSQL documentation.

The following configuration options are supported for GSSAPI:

include_realm

If set to 1, the realm name from the authenticated user principal is included in the system user name that is passed through user name mapping. This is the recommended configuration as, otherwise, it is impossible to differentiate users with the same username who are from different realms. The default for this parameter is 0 (meaning to not include the realm in the system user name) but may change to 1 in a future version of Greenplum Database. You can set it explicitly to avoid any issues when upgrading.

map

Allows for mapping between system and database user names. For a GSSAPI/Kerberos principal, such as `username@EXAMPLE.COM` (or, less commonly, `username/hostbased@EXAMPLE.COM`), the default user name used for mapping is `username` (or `username/hostbased`, respectively), unless `include_realm` has been set to 1 (as recommended, see above), in which case `username@EXAMPLE.COM` (or `username/hostbased@EXAMPLE.COM`) is what is seen as the system username when mapping.

krb_realm

Sets the realm to match user principal names against. If this parameter is set, only users of that realm will be accepted. If it is not set, users of any realm can connect, subject to whatever user name mapping is done.

LDAP Authentication

You can authenticate against an LDAP directory.

- LDAPS and LDAP over TLS options encrypt the connection to the LDAP server.
- The connection from the client to the server is not encrypted unless SSL is enabled. Configure client connections to use SSL to encrypt connections from the client.
- To configure or customize LDAP settings, set the `LDAPCONF` environment variable with the path to the `ldap.conf` file and add this to the `greenplum_path.sh` script.

Following are the recommended steps for configuring your system for LDAP authentication:

1. Set up the LDAP server with the database users/roles to be authenticated via LDAP.
2. On the database:
   a. Verify that the database users to be authenticated via LDAP exist on the database. LDAP is only used for verifying username/password pairs, so the roles should exist in the database.
   b. Update the `pg_hba.conf` file in the `$MASTER_DATA_DIRECTORY` to use LDAP as the authentication method for the respective users. Note that the first entry to match the user/role in the `pg_hba.conf` file will be used as the authentication mechanism, so the position of the entry in the file is important.
   c. Reload the server for the `pg_hba.conf` configuration settings to take effect (`gpstop -u`).

Specify the following parameter `auth-options`.

ldapserver

Names or IP addresses of LDAP servers to connect to. Multiple servers may be specified, separated by spaces.

ldapprefix

String to prepend to the user name when forming the DN to bind as, when doing simple bind authentication.

ldapsuffix

String to append to the user name when forming the DN to bind as, when doing simple bind authentication.

ldapport

Port number on LDAP server to connect to. If no port is specified, the LDAP library's default port setting will be used.

ldaptls

Set to 1 to make the connection between PostgreSQL and the LDAP server use TLS encryption. Note that this only encrypts the traffic to the LDAP server — the connection to the client will still be unencrypted unless SSL is used.

ldapbasedn

Root DN to begin the search for the user in, when doing search+bind authentication.

ldapbinddn

DN of user to bind to the directory with to perform the search when doing search+bind authentication.

ldapbindpasswd

Password for user to bind to the directory with to perform the search when doing search +bind authentication.

ldapsearchattribute

Attribute to match against the user name in the search when doing search+bind authentication.

Example:

```
ldapserver=ldap.greenplum.com prefix="cn=" suffix=", dc=greenplum, dc=com"
```

Following are sample `pg_hba.conf` file entries for LDAP authentication:

```
host all testuser 0.0.0.0/0 ldap ldap
ldapserver=ldapserver.greenplum.com ldapport=389 ldapprefix="cn="
 ldapsuffix=",ou=people,dc=greenplum,dc=com"
hostssl   all   ldaprole   0.0.0.0/0   ldap
ldapserver=ldapserver.greenplum.com ldaptls=1 ldapprefix="cn="
 ldapsuffix=",ou=people,dc=greenplum,dc=com"
```

SSL Client Authentication

SSL authentication compares the Common Name (cn) attribute of an SSL certificate provided by the connecting client during the SSL handshake to the requested database user name. The database user should exist in the database. A map file can be used for mapping between system and database user names.

SSL Authentication Parameters

Authentication method:

- Cert

  Authentication options:
  Hostssl

  Connection type must be hostssl.

  map=*mapping*

  mapping.

  This is specified in the `pg_ident.conf`, or in the file specified in the `ident_file` server setting.

Following are sample `pg_hba.conf` entries for SSL client authentication:

```
Hostssl testdb certuser 192.168.0.0/16 cert
Hostssl testdb all 192.168.0.0/16 cert map=gpuser
```

OpenSSL Configuration

Greenplum Database reads the OpenSSL configuration file specified in `$GP_HOME/etc/openssl.cnf` by default. You can make changes to the default configuration for OpenSSL by modifying or updating this file and restarting the server.

Creating a Self-Signed Certificate

A self-signed certificate can be used for testing, but a certificate signed by a certificate authority (CA) (either one of the global CAs or a local one) should be used in production so that clients can verify the server's identity. If all the clients are local to the organization, using a local CA is recommended.

To create a self-signed certificate for the server:

1. Enter the following `openssl` command:

   ```
   openssl req -new -text -out server.req
   ```

2. Enter the requested information at the prompts.

   Make sure you enter the local host name for the Common Name. The challenge password can be left blank.

3. The program generates a key that is passphrase-protected; it does not accept a passphrase that is less than four characters long. To remove the passphrase (and you must if you want automatic start-up of the server), run the following command:

   ```
   openssl rsa -in privkey.pem -out server.key rm privkey.pem
   ```

4. Enter the old passphrase to unlock the existing key. Then run the following command:

   ```
   openssl req -x509 -in server.req -text -key server.key -out server.crt
   ```

   This turns the certificate into a self-signed certificate and copies the key and certificate to where the server will look for them.

5. Finally, run the following command:

   ```
   chmod og-rwx server.key
   ```

For more details on how to create your server private key and certificate, refer to the OpenSSL documentation.

Configuring postgresql.conf for SSL Authentication

The following Server settings need to be specified in the `postgresql.conf` configuration file:

- `ssl` *boolean*. Enables SSL connections.
- `ssl_renegotiation_limit` *integer*. Specifies the data limit before key renegotiation.
- `ssl_ciphers` *string*. Configures the list SSL ciphers that are allowed. `ssl_ciphers` *overrides* any ciphers string specified in `/etc/openssl.cnf`. The default value `ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH` enables all ciphers except for ADH, LOW, EXP, and MD5 ciphers, and prioritizes ciphers by their strength.

Note: With TLS 1.2 some ciphers in MEDIUM and HIGH strength still use NULL encryption (no encryption for transport), which the default `ssl_ciphers` string allows. To bypass NULL ciphers with TLS 1.2 use a string such as `TLSv1.2:!eNULL:!aNULL`.

It is possible to have authentication without encryption overhead by using `NULL-SHA` or `NULL-MD5` ciphers. However, a man-in-the-middle could read and pass communications between client and server. Also, encryption overhead is minimal compared to the overhead of authentication. For these reasons, NULL ciphers should not be used.

The default location for the following SSL server files is the Greenplum Database master data directory (`$MASTER_DATA_DIRECTORY`):

- `server.crt` - Server certificate.
- `server.key` - Server private key.
- `root.crt` - Trusted certificate authorities.
- `root.crl` - Certificates revoked by certificate authorities.

If Greenplum Database master mirroring is enabled with SSL client authentication, the SSL server files *should not be placed* in the default directory `$MASTER_DATA_DIRECTORY`. If an `initstandby` operation is performed, the contents of `$MASTER_DATA_DIRECTORY` is copied from the master to the standby master and the incorrect SSL key, and cert files (the master files, and not the standby master files) will prevent standby master start up.

You can specify a different directory for the location of the SSL server files with the `postgresql.conf` parameters `sslcert`, `sslkey`, `sslrootcert`, and `sslcrl`.

Configuring the SSL Client Connection

SSL options:

`sslmode`

Specifies the level of protection.

`require`

Only use an SSL connection. If a root CA file is present, verify the certificate in the same way as if `verify-ca` was specified.

`verify-ca`

Only use an SSL connection. Verify that the server certificate is issued by a trusted CA.

`verify-full`

Only use an SSL connection. Verify that the server certificate is issued by a trusted CA and that the server host name matches that in the certificate.

`sslcert`

The file name of the client SSL certificate. The default is `$MASTER_DATA_DIRECTORY/postgresql.crt`.

`sslkey`

The secret key used for the client certificate. The default is `$MASTER_DATA_DIRECTORY/postgresql.key`.

`sslrootcert`

The name of a file containing SSL Certificate Authority certificate(s). The default is `$MASTER_DATA_DIRECTORY/root.crt`.

`sslcrl`

The name of the SSL certificate revocation list. The default is `$MASTER_DATA_DIRECTORY/root.crl`.

The client connection parameters can be set using the following environment variables:

- `sslmode` – `PGSSLMODE`
- `sslcert` – `PGSSLCERT`
- `sslkey` – `PGSSLKEY`
- `sslrootcert` – `PGSSLROOTCERT`
- `sslcrl` – `PGSSLCRL`

PAM-Based Authentication

The "PAM" (Pluggable Authentication Modules) authentication method validates username/password pairs, similar to basic authentication. To use PAM authentication, the user must already exist as a Greenplum Database role name.

Greenplum uses the `pamservice` authentication parameter to identify the service from which to obtain the PAM configuration.

> **Note:** If PAM is set up to read `/etc/shadow`, authentication will fail because the PostgreSQL server is started by a non-root user. This is not an issue when PAM is configured to use LDAP or another authentication method.

Greenplum Database does not install a PAM configuration file. If you choose to use PAM authentication with Greenplum, you must identify the PAM service name for Greenplum and create the associated PAM service configuration file and configure Greenplum Database to use PAM authentication as described below:

1. Log in to the Greenplum Database master host and set up your environment. For example:

```
$ ssh gpadmin@<gpmaster>
gpadmin@gpmaster$ . /usr/local/greenplum-db/greenplum_path.sh
```

2. Identify the `pamservice` name for Greenplum Database. In this procedure, we choose the name `greenplum`.
3. Create the PAM service configuration file, `/etc/pam.d/greenplum`, and add the text below. You must have operating system superuser privileges to create the `/etc/pam.d` directory (if necessary) and the `greenplum` PAM configuration file.

```
#%PAM-1.0
auth  include  password-auth
account  include  password-auth
```

   This configuration instructs PAM to authenticate the local operating system user.
4. Ensure that the `/etc/pam.d/greenplum` file is readable by all users:

```
sudo chmod 644 /etc/pam.d/greenplum
```

5. Add one or more entries to the `pg_hba.conf` configuration file to enable PAM authentication in Greenplum Database. These entries must specify the `pam` *auth-method*. You must also specify the `pamservice=greenplum` *auth-option*. For example:

```

host     <user-name>     <db-name>     <address>     pam
 pamservice=greenplum
```

6. Reload the Greenplum Database configuration:

```
$ gpstop -u
```

Radius Authentication

RADIUS (Remote Authentication Dial In User Service) authentication works by sending an Access Request message of type 'Authenticate Only' to a configured RADIUS server. It includes parameters for user name, password (encrypted), and the Network Access Server (NAS) Identifier. The request is encrypted using the shared secret specified in the `radiussecret` option. The RADIUS server responds with either `Access Accept` or `Access Reject`.

Note: RADIUS accounting is not supported.

RADIUS authentication only works if the users already exist in the database.

The RADIUS encryption vector requires SSL to be enabled in order to be cryptographically strong.

RADIUS Authentication Options

`radiusserver`

The name of the RADIUS server.

`radiussecret`

The RADIUS shared secret.

`radiusport`

The port to connect to on the RADIUS server.

`radiusidentifier`

NAS identifier in RADIUS requests.

Following are sample `pg_hba.conf` entries for RADIUS client authentication:

```
hostssl  all all 0.0.0.0/0 radius radiusserver=servername
 radiussecret=sharedsecret
```

Limiting Concurrent Connections

To limit the number of active concurrent sessions to your Greenplum Database system, you can configure the `max_connections` server configuration parameter. This is a local parameter, meaning that you must set it in the `postgresql.conf` file of the master, the standby master, and each segment instance (primary and mirror). The value of `max_connections` on segments must be 5-10 times the value on the master.

When you set `max_connections`, you must also set the dependent parameter `max_prepared_transactions`. This value must be at least as large as the value of `max_connections` on the master, and segment instances should be set to the same value as the master.

In `$MASTER_DATA_DIRECTORY/postgresql.conf` (including standby master):

```
max_connections=100
max_prepared_transactions=100
```

In `SEGMENT_DATA_DIRECTORY/postgresql.conf` for all segment instances:

```
max_connections=500
max_prepared_transactions=100
```

Note: Note: Raising the values of these parameters may cause Greenplum Database to request more shared memory. To mitigate this effect, consider decreasing other memory-related parameters such as `gp_cached_segworkers_threshold`.

To change the number of allowed connections:

1. Stop your Greenplum Database system:

```
$ gpstop
```

2. On the master host, edit `$MASTER_DATA_DIRECTORY/postgresql.conf` and change the following two parameters:
   - `max_connections` – the number of active user sessions you want to allow plus the number of `superuser_reserved_connections`.
   - `max_prepared_transactions` – must be greater than or equal to `max_connections`.
3. On each segment instance, edit `SEGMENT_DATA_DIRECTORY/postgresql.conf` and change the following two parameters:
   - `max_connections` – must be 5-10 times the value on the master.
   - `max_prepared_transactions` – must be equal to the value on the master.
4. Restart your Greenplum Database system:

```
$ gpstart
```

Encrypting Client/Server Connections

Greenplum Database has native support for SSL connections between the client and the master server. SSL connections prevent third parties from snooping on the packets, and also prevent man-in-the-middle attacks. SSL should be used whenever the client connection goes through an insecure link, and must be used whenever client certificate authentication is used.

> **Note:** For information about encrypting data between the `gpfdist` server and Greenplum Database segment hosts, see *Encrypting gpfdist Connections*.

To enable SSL requires that OpenSSL be installed on both the client and the master server systems. Greenplum can be started with SSL enabled by setting the server configuration parameter `ssl=on` in the master `postgresql.conf`. When starting in SSL mode, the server will look for the files `server.key` (server private key) and `server.crt` (server certificate) in the master data directory. These files must be set up correctly before an SSL-enabled Greenplum system can start.

> **Important:** Do not protect the private key with a passphrase. The server does not prompt for a passphrase for the private key, and the database startup fails with an error if one is required.

A self-signed certificate can be used for testing, but a certificate signed by a certificate authority (CA) should be used in production, so the client can verify the identity of the server. Either a global or local CA can be used. If all the clients are local to the organization, a local CA is recommended. See *Creating a Self-Signed Certificate* for steps to create a self-signed certificate.

Configuring Database Authorization

Describes how to restrict authorization access to database data at the user level by using roles and permissions.

Access Permissions and Roles

Greenplum Database manages database access permissions using *roles*. The concept of roles subsumes the concepts of users and groups. A role can be a database user, a group, or both. Roles can own database objects (for example, tables) and can assign privileges on those objects to other roles to control access to the objects. Roles can be members of other roles, thus a member role can inherit the object privileges of its parent role.

Every Greenplum Database system contains a set of database roles (users and groups). Those roles are separate from the users and groups managed by the operating system on which the server runs. However, for convenience you may want to maintain a relationship between operating system user names and Greenplum Database role names, since many of the client applications use the current operating system user name as the default.

In Greenplum Database, users log in and connect through the master instance, which verifies their role and access privileges. The master then issues out commands to the segment instances behind the scenes using the currently logged in role.

Roles are defined at the system level, so they are valid for all databases in the system.

To bootstrap the Greenplum Database system, a freshly initialized system always contains one predefined superuser role (also referred to as the system user). This role will have the same name as the operating system user that initialized the Greenplum Database system. Customarily, this role is named `gpadmin`. To create more roles you first must connect as this initial role.

Managing Object Privileges

When an object (table, view, sequence, database, function, language, schema, or tablespace) is created, it is assigned an owner. The owner is normally the role that executed the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, privileges must be granted. Greenplum Database supports the following privileges for each object type:

Object Type	Privileges

Tables, Views, Sequences	`SELECT`
	`INSERT`
	`UPDATE`
	`DELETE`
	`RULE`
	`ALL`

External Tables	`SELECT`
	`RULE`
	`ALL`

Object Type	Privileges
Databases	CONNECT CREATE TEMPORARY | TEMP ALL
Functions	EXECUTE
Procedural Languages	USAGE
Schemas	CREATE USAGE ALL

Privileges must be granted for each object individually. For example, granting `ALL` on a database does not grant full access to the objects within that database. It only grants all of the database-level privileges (`CONNECT`, `CREATE`, `TEMPORARY`) to the database itself.

Use the `GRANT` SQL command to give a specified role privileges on an object. For example:

```
=# GRANT INSERT ON mytable TO jsmith;
```

To revoke privileges, use the `REVOKE` command. For example:

```
=# REVOKE ALL PRIVILEGES ON mytable FROM jsmith;
```

You can also use the `DROP OWNED` and `REASSIGN OWNED` commands for managing objects owned by deprecated roles. (Note: only an object's owner or a superuser can drop an object or reassign ownership.) For example:

```
=# REASSIGN OWNED BY sally TO bob;
=# DROP OWNED BY visitor;
```

Using SSH-256 Encryption

Greenplum Database access control corresponds roughly to the Orange Book 'C2' level of security, not the 'B1' level. Greenplum Database currently supports access privileges at the object level. Greenplum Database does not support row-level access or row-level, labeled security.

You can simulate row-level access by using views to restrict the rows that are selected. You can simulate row-level labels by adding an extra column to the table to store sensitivity information, and then using views to control row-level access based on this column. You can then grant roles access to the views rather than the base table. While these workarounds do not provide the same as "B1" level security, they may still be a viable alternative for many organizations.

To use SHA-256 encryption, you must set a parameter either at the system or the session level. This section outlines how to use a server parameter to implement SHA-256 encrypted password storage. Note that in order to use SHA-256 encryption for storage, the client authentication method must be set to password rather than the default, MD5. (See *Configuring the SSL Client Connection* for more details.) This means that the password is transmitted in clear text over the network, so we highly recommend that you set up SSL to encrypt the client server communication channel.

You can set your chosen encryption method system-wide or on a per-session basis. The available encryption methods are SHA-256 and MD5 (for backward compatibility).

Setting Encryption Method System-wide

To set the `password_hash_algorithm` server parameter on a complete Greenplum system (master and its segments):

1. Log in to your Greenplum Database instance as a superuser.
2. Execute `gpconfig` with the `password_hash_algorithm` set to SHA-256:

```
$ gpconfig -c password_hash_algorithm -v 'SHA-256'
```

3. Verify the setting:

```
$ gpconfig -s
```

You will see:

```
Master value: SHA-256
Segment value: SHA-256
```

Setting Encryption Method for an Individual Session

To set the `password_hash_algorithm` server parameter for an individual session:

1. Log in to your Greenplum Database instance as a superuser.
2. Set the `password_hash_algorithm` to SHA-256:

```
# set password_hash_algorithm = 'SHA-256'
```

3. Verify the setting:

```
# show password_hash_algorithm;
```

You will see:

```
SHA-256
```

Following is an example of how the new setting works:

1. Log in as a super user and verify the password hash algorithm setting:

```
# show password_hash_algorithm
 password_hash_algorithm
 -------------------------------
 SHA-256
```

2. Create a new role with password that has login privileges.

```
create role testdb with password 'testdb12345#' LOGIN;
```

3. Change the client authentication method to allow for storage of SHA-256 encrypted passwords:

   Open the `pg_hba.conf` file on the master and add the following line:

```
host all testdb 0.0.0.0/0 password
```

4. Restart the cluster.

5. Log in to the database as the user just created, `testdb`.

```
psql -U testdb
```

6. Enter the correct password at the prompt.
7. Verify that the password is stored as a SHA-256 hash.

   Password hashes are stored in `pg_authid.rolpasswod`.
8. Log in as the super user.
9. Execute the following query:

```
# SELECT rolpassword FROM pg_authid WHERE rolname = 'testdb';
Rolpassword
-----------
sha256<64 hexadecimal characters>
```

Restricting Access by Time

Greenplum Database enables the administrator to restrict access to certain times by role. Use the `CREATE ROLE` or `ALTER ROLE` commands to specify time-based constraints.

Access can be restricted by day or by day and time. The constraints are removable without deleting and recreating the role.

Time-based constraints only apply to the role to which they are assigned. If a role is a member of another role that contains a time constraint, the time constraint is not inherited.

Time-based constraints are enforced only during login. The `SET ROLE` and `SET SESSION AUTHORIZATION` commands are not affected by any time-based constraints.

Superuser or `CREATEROLE` privileges are required to set time-based constraints for a role. No one can add time-based constraints to a superuser.

There are two ways to add time-based constraints. Use the keyword `DENY` in the `CREATE ROLE` or `ALTER ROLE` command followed by one of the following.

- A day, and optionally a time, when access is restricted. For example, no access on Wednesdays.
- An interval—that is, a beginning and ending day and optional time—when access is restricted. For example, no access from Wednesday 10 p.m. through Thursday at 8 a.m.

You can specify more than one restriction; for example, no access Wednesdays at any time and no access on Fridays between 3:00 p.m. and 5:00 p.m.

There are two ways to specify a day. Use the word `DAY` followed by either the English term for the weekday, in single quotation marks, or a number between 0 and 6, as shown in the table below.

English Term	Number
DAY 'Sunday'	DAY 0
DAY 'Monday'	DAY 1
DAY 'Tuesday'	DAY 2
DAY 'Wednesday'	DAY 3
DAY 'Thursday'	DAY 4
DAY 'Friday'	DAY 5
DAY 'Saturday'	DAY 6

A time of day is specified in either 12- or 24-hour format. The word `TIME` is followed by the specification in single quotation marks. Only hours and minutes are specified and are separated by a colon (:). If using a 12-hour format, add `AM` or `PM` at the end. The following examples show various time specifications.

```
TIME '14:00'     # 24-hour time implied
TIME '02:00 PM'  # 12-hour time specified by PM
TIME '02:00'     # 24-hour time implied. This is equivalent to TIME '02:00
 AM'.
```

> **Important:** Time-based authentication is enforced with the server time. Timezones are disregarded.

To specify an interval of time during which access is denied, use two day/time specifications with the words `BETWEEN` and `AND`, as shown. `DAY` is always required.

```
BETWEEN DAY 'Monday' AND DAY 'Tuesday'

BETWEEN DAY 'Monday' TIME '00:00' AND
        DAY 'Monday' TIME '01:00'

BETWEEN DAY 'Monday' TIME '12:00 AM' AND
        DAY 'Tuesday' TIME '02:00 AM'

BETWEEN DAY 'Monday' TIME '00:00' AND
        DAY 'Tuesday' TIME '02:00'
        DAY 2 TIME '02:00'
```

The last three statements are equivalent.

> **Note:** Intervals of days cannot wrap past Saturday.

The following syntax is not correct:

```
DENY BETWEEN DAY 'Saturday' AND DAY 'Sunday'
```

The correct specification uses two DENY clauses, as follows:

```
DENY DAY 'Saturday'
DENY DAY 'Sunday'
```

The following examples demonstrate creating a role with time-based constraints and modifying a role to add time-based constraints. Only the statements needed for time-based constraints are shown. For more details on creating and altering roles see the descriptions of `CREATE ROLE` and `ALTER ROLE` in the *Greenplum Database Reference Guide*.

Example 1 – Create a New Role with Time-based Constraints

No access is allowed on weekends.

```
 CREATE ROLE generaluser
 DENY DAY 'Saturday'
 DENY DAY 'Sunday'
 ...
```

Example 2 – Alter a Role to Add Time-based Constraints

No access is allowed every night between 2:00 a.m. and 4:00 a.m.

```
ALTER ROLE generaluser
 DENY BETWEEN DAY 'Monday' TIME '02:00' AND DAY 'Monday' TIME '04:00'
```

```
 DENY BETWEEN DAY 'Tuesday' TIME '02:00' AND DAY 'Tuesday' TIME '04:00'
 DENY BETWEEN DAY 'Wednesday' TIME '02:00' AND DAY 'Wednesday' TIME '04:00'
 DENY BETWEEN DAY 'Thursday' TIME '02:00' AND DAY 'Thursday' TIME '04:00'
 DENY BETWEEN DAY 'Friday' TIME '02:00' AND DAY 'Friday' TIME '04:00'
 DENY BETWEEN DAY 'Saturday' TIME '02:00' AND DAY 'Saturday' TIME '04:00'
 DENY BETWEEN DAY 'Sunday' TIME '02:00' AND DAY 'Sunday' TIME '04:00'
  ...
```

Excample 3 – Alter a Role to Add Time-based Constraints

No access is allowed Wednesdays or Fridays between 3:00 p.m. and 5:00 p.m.

```
ALTER ROLE generaluser
 DENY DAY 'Wednesday'
 DENY BETWEEN DAY 'Friday' TIME '15:00' AND DAY 'Friday' TIME '17:00'
```

Dropping a Time-based Restriction

To remove a time-based restriction, use the ALTER ROLE command. Enter the keywords DROP DENY FOR followed by a day/time specification to drop.

```
DROP DENY FOR DAY 'Sunday'
```

Any constraint containing all or part of the conditions in a DROP clause is removed. For example, if an existing constraint denies access on Mondays and Tuesdays, and the DROP clause removes constraints for Mondays, the existing constraint is completely dropped. The DROP clause completely removes all constraints that overlap with the constraint in the drop clause. The overlapping constraints are completely removed even if they contain more restrictions that the restrictions mentioned in the DROP clause.

Example 1 - Remove a Time-based Restriction from a Role

```
 ALTER ROLE generaluser
 DROP DENY FOR DAY 'Monday'
    ...
```

This statement would remove all constraints that overlap with a Monday constraint for the role `generaluser` in Example 2, even if there are additional constraints.

Greenplum Command Center Security

Greenplum Command Center is a web-based application for monitoring and managing Greenplum clusters. Command Center works with data collected by agents running on the segment hosts and saved to the gpperfmon database. Installing Command Center creates the gpperfmon database and the gpmon database role if they do not already exist. It creates the gpmetrics schema in the gpperfmon database, which contains metrics and query history tables populated by the Greenplum Database metrics collector module.

> **Note:** The gpperfmon_install utility also creates the gpperfmon database and gpmon role, but Command Center no longer requires the history tables it creates in the database. Do not use gpperfmon_install unless you need the old query history tables for some other purpose. gpperfmon_install enables the gpmmon and gpsmon agents, which add unnecessary load to the Greenplum Database system if you do not need the old history tables.

The gpmon User

The Command Center installer creates the gpmon database role and adds the role to the pg_hba.conf file with the following entries:

```
local    gpperfmon   gpmon         md5
host     all         gpmon         127.0.0.1/28    md5
host     all         gpmon         ::1/128         md5
```

These entries allow gpmon to establish a local socket connection to the gpperfmon database and a TCP/IP connection to any database.

The gpmon database role is a superuser. In a secure or production environment, it may be desirable to restrict the gpmon user to just the gpperfmon database. Do this by editing the gpmon host entry in the pg_hba.conf file and changing all in the database field to gpperfmon:

```
local   gpperfmon   gpmon                       md5
host    gpperfmon   gpmon    127.0.0.1/28       md5
host    gpperfmon   gpmon    ::1/128            md5
```

The password used to authenticate the gpmon user is stored in the gpadmin home directory in the ~/.pgpass file. The ~/.pgpass file must be owned by the gpadmin user and be RW-accessible only by the gpadmin user. The Command Center installer creates the gpmon role with the default password "changeme". Be sure to change the password immediately after you install Command Center. Use the ALTER ROLE command to change the password in the database, change the password in the ~/.pgpass file, and then restart Command Center with the gpcc start command.

Because the .pgpass file contains the plain-text password of the gpmon user, you may want to remove it and supply the gpmon password using a more secure method. The gpmon password is needed when you run the gpcc start, gpcc stop, or gpcc status commands. You can add the -W option to the gpcc command to have the command prompt you to enter the password. Alternatively, you can set the PGPASSWORD environment variable to the gpmon password before you run the gpcc command.

Command Center does not allow logins from any role configured with trust authentication, including the gpadmin user.

The gpmon user can log in to the Command Center Console and has access to all of the application's features. You can allow other database roles access to Command Center so that you can secure the gpmon user and restrict other users' access to Command Center features. Setting up other Command Center users is described in the next section.

Greenplum Command Center Users

To log in to the Command Center web application, a user must be allowed access to the gpperfmon database in `pg_hba.conf`. For example, to make `user1` a regular Command Center user, edit the `pg_hba.conf` file and either add or edit a line for the user so that the gpperfmon database is included in the database field. For example:

```
host     gpperfmon,accounts   user1     127.0.0.1/28    md5
```

The Command Center web application includes an Admin interface to add, remove, and edit entries in the `pg_hba.conf` file and reload the file into Greenplum Database.

Command Center has the following types of users:

- *Self Only* users can view metrics and view and cancel their own queries. Any Greenplum Database user successfully authenticated through the Greenplum Database authentication system can access Greenplum Command Center with Self Only permission. Higher permission levels are required to view and cancel other's queries and to access the System and Admin Control Center features.
- *Basic* users can view metrics, view all queries, and cancel their own queries. Users with Basic permission are members of the Greenplum Database `gpcc_basic` group.
- *Operator Basic* users can view metrics, view their own and others' queries, cancel their own queries, and view the System and Admin screens. Users with Operator Basic permission are members of the Greenplum Database `gpcc_operator_basic` group.
- *Operator* users can view their own and others' queries, cancel their own and other's queries, and view the System and Admin screens. Users with Operator permission are members of the Greenplum Database `gpcc_operator` group.
- *Admin* users can access all views and capabilities in the Command Center. Greenplum Database users with the SUPERUSER privilege have Admin permissions in Command Center.

The Command Center web application has an Admin interface you can use to change a Command Center user's access level.

Enabling SSL for Greenplum Command Center

The Command Center web server can be configured to support SSL so that client connections are encrypted. To enable SSL, install a `.pem` file containing the web server's certificate and private key on the web server host and then enter the full path to the `.pem` file when prompted by the Command Center installer.

Enabling Kerberos Authentication for Greenplum Command Center Users

If Kerberos authentication is enabled for Greenplum Database, Command Center users can also authenticate with Kerberos. Command Center supports three Kerberos authentication modes: *strict*, *normal*, and *gpmon-only*.

`Strict`

Command Center has a Kerberos keytab file containing the Command Center service principal and a principal for every Command Center user. If the principal in the client's connection request is in the keytab file, the web server grants the client access and the web server connects to Greenplum Database using the client's principal name. If the principal is not in the keytab file, the connection request fails.

`Normal`

The Command Center Kerberos keytab file contains the Command Center principal and may contain principals for Command Center users. If the principal in the client's connection request is in Command Center's keytab file, it uses the client's principal for database connections. Otherwise, Command Center uses the `gpmon` user for database connections.

`gpmon-only`

The Command Center uses the `gpmon` database role for all Greenplum Database connections. No client principals are needed in the Command Center's keytab file.

See the *Greenplum Command Center documentation* for instructions to enable Kerberos authentication with Greenplum Command Center

Auditing

Describes Greenplum Database events that are logged and should be monitored to detect security threats.

Greenplum Database is capable of auditing a variety of events, including startup and shutdown of the system, segment database failures, SQL statements that result in an error, and all connection attempts and disconnections. Greenplum Database also logs SQL statements and information regarding SQL statements, and can be configured in a variety of ways to record audit information with more or less detail. The `log_error_verbosity` configuration parameter controls the amount of detail written in the server log for each message that is logged. Similarly, the `log_min_error_statement` parameter allows administrators to configure the level of detail recorded specifically for SQL statements, and the `log_statement` parameter determines the kind of SQL statements that are audited. Greenplum Database records the username for all auditable events, when the event is initiated by a subject outside the Greenplum Database.

Greenplum Database prevents unauthorized modification and deletion of audit records by only allowing administrators with an appropriate role to perform any operations on log files. Logs are stored in a proprietary format using comma-separated values (CSV). Each segment and the master stores its own log files, although these can be accessed remotely by an administrator. Greenplum Database also authorizes overwriting of old log files via the `log_truncate_on_rotation` parameter. This is a local parameter and must be set on each segment and master configuration file.

Greenplum provides an administrative schema called `gp_toolkit` that you can use to query log files, as well as system catalogs and operating environment for system status information. For more information, including usage, refer to *The gp_tookit Administrative Schema* appendix in the *Greenplum Database Reference Guide*.

Viewing the Database Server Log Files

Every database instance in Greenplum Database (master and segments) is a running PostgreSQL database server with its own server log file. Daily log files are created in the `pg_log` directory of the master and each segment data directory.

The server log files are written in comma-separated values (CSV) format. Not all log entries will have values for all of the log fields. For example, only log entries associated with a query worker process will have the `slice_id` populated. Related log entries of a particular query can be identified by its session identifier (`gp_session_id`) and command identifier (`gp_command_count`).

#	Field Name	Data Type	Description
1	event_time	timestamp with time zone	Time that the log entry was written to the log
2	user_name	varchar(100)	The database user name
3	database_name	varchar(100)	The database name
4	process_id	varchar(10)	The system process id (prefixed with "p")
5	thread_id	varchar(50)	The thread count (prefixed with "th")
6	remote_host	varchar(100)	On the master, the hostname/address of the client machine. On the segment, the hostname/address of the master.

#	Field Name	Data Type	Description
7	remote_port	varchar(10)	The segment or master port number
8	session_start_time	timestamp with time zone	Time session connection was opened
9	transaction_id	int	Top-level transaction ID on the master. This ID is the parent of any subtransactions.
10	gp_session_id	text	Session identifier number (prefixed with "con")
11	gp_command_count	text	The command number within a session (prefixed with "cmd")
12	gp_segment	text	The segment content identifier (prefixed with "seg" for primaries or "mir" for mirrors). The master always has a content id of -1.
13	slice_id	text	The slice id (portion of the query plan being executed)
14	distr_tranx_id	text	Distributed transaction ID
15	local_tranx_id	text	Local transaction ID
16	sub_tranx_id	text	Subtransaction ID
17	event_severity	varchar(10)	Values include: LOG, ERROR, FATAL, PANIC, DEBUG1, DEBUG2
18	sql_state_code	varchar(10)	SQL state code associated with the log message
19	event_message	text	Log or error message text
20	event_detail	text	Detail message text associated with an error or warning message
21	event_hint	text	Hint message text associated with an error or warning message
22	internal_query	text	The internally-generated query text
23	internal_query_pos	int	The cursor index into the internally-generated query text
24	event_context	text	The context in which this message gets generated

#	Field Name	Data Type	Description
25	debug_query_string	text	User-supplied query string with full detail for debugging. This string can be modified for internal use.
26	error_cursor_pos	int	The cursor index into the query string
27	func_name	text	The function in which this message is generated
28	file_name	text	The internal code file where the message originated
29	file_line	int	The line of the code file where the message originated
30	stack_trace	text	Stack trace text associated with this message

Greenplum provides a utility called `gplogfilter` that can be used to search through a Greenplum Database log file for entries matching the specified criteria. By default, this utility searches through the Greenplum master log file in the default logging location. For example, to display the last three lines of the master log file:

```
$ gplogfilter -n 3
```

You can also use `gplogfilter` to search through all segment log files at once by running it through the `gpssh` utility. For example, to display the last three lines of each segment log file:

```
$ gpssh -f seg_host_file
  => source /usr/local/greenplum-db/greenplum_path.sh
  => gplogfilter -n 3 /gpdata/gp*/pg_log/gpdb*.csv
```

The following are the Greenplum security-related audit (or logging) server configuration parameters that are set in the postgresql.conf configuration file:

Field Name	Value Range	Default	Description
log_connections	Boolean	off	This outputs a line to the server log detailing each successful connection. Some client programs, like psql, attempt to connect twice while determining if a password is required, so duplicate “connection received” messages do not always indicate a problem.
log_disconnections	Boolean	off	This outputs a line in the server log at termination of a client session, and includes the duration of the session.

Field Name	Value Range	Default	Description
log_statement	NONE DDL MOD ALL	ALL	Controls which SQL statements are logged. DDL logs all data definition commands like CREATE, ALTER, and DROP commands. MOD logs all DDL statements, plus INSERT, UPDATE, DELETE, TRUNCATE, and COPY FROM. PREPARE and EXPLAIN ANALYZE statements are also logged if their contained command is of an appropriate type.
log_hostname	Boolean	off	By default, connection log messages only show the IP address of the connecting host. Turning on this option causes logging of the host name as well. Note that depending on your host name resolution setup this might impose a non-negligible performance penalty.
log_duration	Boolean	off	Causes the duration of every completed statement which satisfies log_statement to be logged.
log_error_verbosity	TERSE DEFAULT VERBOSE	DEFAULT	Controls the amount of detail written in the server log for each message that is logged.
log_min_duration_ statement	number of milliseconds, 0, -1	-1	Logs the statement and its duration on a single log line if its duration is greater than or equal to the specified number of milliseconds. Setting this to 0 will print all statements and their durations. -1 disables the feature. For example, if you set it to 250 then all SQL statements that run 250ms or longer will be logged. Enabling this option can be useful in tracking down unoptimized queries in your applications.

Field Name	Value Range	Default	Description
log_min_messages	DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 INFO NOTICE WARNING ERROR LOG FATAL PANIC	NOTICE	Controls which message levels are written to the server log. Each level includes all the levels that follow it. The later the level, the fewer messages are sent to the log.
log_rotation_size	0 - INT_MAX/1024 kilobytes	1048576	When greater than 0, a new log file is created when this number of kilobytes have been written to the log. Set to zero to disable size-based creation of new log files.
log_rotation_age	Any valid time expression (number and unit)	1d	Determines the lifetime of an individual log file. When this amount of time has elapsed since the current log file was created, a new log file will be created. Set to zero to disable time-based creation of new log files.
log_statement_stats	Boolean	off	For each query, write total performance statistics of the query parser, planner, and executor to the server log. This is a crude profiling instrument.
log_truncate_on_ rotation	Boolean	off	Truncates (overwrites), rather than appends to, any existing log file of the same name. Truncation will occur only when a new file is being opened due to time-based rotation. For example, using this setting in combination with a log_filename such as gpseg#-%H.log would result in generating twenty-four hourly log files and then cyclically overwriting them. When off, pre-existing files will be appended to in all cases.

Encrypting Data and Database Connections

Describes how to encrypt data at rest in the database or in transit over the network, to protect from evesdroppers or man-in-the-middle attacks.

- Connections between clients and the master database can be encrypted with SSL. This is enabled with the `ssl` server configuration parameter, which is `off` by default. Setting the `ssl` parameter to `on` allows client communications with the master to be encrypted. The master database must be set up for SSL. See *OpenSSL Configuration* for more about encrypting client connections with SSL.
- Greenplum Database allows SSL encryption of data in transit between the Greenplum parallel file distribution server, `gpfdist`, and segment hosts. See *Encrypting gpfdist Connections* for more information.
- The `pgcrypto` module of encryption/decryptions functions protect data at rest in the database. Encryption at the column level protects sensitive information, such as social security numbers or credit card numbers. See *Encrypting Data at Rest with pgcrypto* for more information.

Encrypting gpfdist Connections

The `gpfdists` protocol is a secure version of the `gpfdist` protocol that securely identifies the file server and the Greenplum Database and encrypts the communications between them. Using `gpfdists` protects against eavesdropping and man-in-the-middle attacks.

The `gpfdists` protocol implements client/server SSL security with the following notable features:

- Client certificates are required.
- Multilingual certificates are not supported.
- A Certificate Revocation List (CRL) is not supported.
- The TLSv1 protocol is used with the `TLS_RSA_WITH_AES_128_CBC_SHA` encryption algorithm. These SSL parameters cannot be changed.
- SSL renegotiation is supported.
- The SSL ignore host mismatch parameter is set to false.
- Private keys containing a passphrase are not supported for the `gpfdist` file server (server.key) or for the Greenplum Database (client.key).
- It is the user's responsibility to issue certificates that are appropriate for the operating system in use. Generally, converting certificates to the required format is supported, for example using the SSL Converter at *https://www.sslshopper.com/ssl-converter.html*.

A `gpfdist` server started with the `--ssl` option can only communicate with the `gpfdists` protocol. A `gpfdist` server started without the `--ssl` option can only communicate with the `gpfdist` protocol. For more detail about `gpfdist` refer to the *Greenplum Database Administrator Guide*.

There are two ways to enable the `gpfdists` protocol:

- Run `gpfdist` with the `--ssl` option and then use the `gpfdists` protocol in the `LOCATION` clause of a `CREATE EXTERNAL TABLE` statement.
- Use a YAML control file with the SSL option set to true and run `gpload`. Running `gpload` starts the `gpfdist` server with the `--ssl` option and then uses the `gpfdists` protocol.

When using gpfdists, the following client certificates must be located in the `$PGDATA/gpfdists` directory on each segment:

- The client certificate file, `client.crt`
- The client private key file, `client.key`
- The trusted certificate authorities, `root.crt`

> **Important:** Do not protect the private key with a passphrase. The server does not prompt for a passphrase for the private key, and loading data fails with an error if one is required.

When using `gpload` with SSL you specify the location of the server certificates in the YAML control file. When using `gpfdist` with SSL, you specify the location of the server certificates with the --ssl option.

The following example shows how to securely load data into an external table. The example creates a readable external table named `ext_expenses` from all files with the `txt` extension, using the `gpfdists` protocol. The files are formatted with a pipe (`|`) as the column delimiter and an empty space as null.

1. Run `gpfdist` with the `--ssl` option on the segment hosts.
2. Log into the database and execute the following command:

```
=# CREATE EXTERNAL TABLE ext_expenses
   ( name text, date date, amount float4, category text, desc1 text )
LOCATION ('gpfdists://etlhost-1:8081/*.txt', 'gpfdists://etlhost-2:8082/
*.txt')
FORMAT 'TEXT' ( DELIMITER '|' NULL ' ') ;
```

Encrypting Data at Rest with pgcrypto

The pgcrypto module for Greenplum Database provides functions for encrypting data at rest in the database. Administrators can encrypt columns with sensitive information, such as social security numbers or credit card numbers, to provide an extra layer of protection. Database data stored in encrypted form cannot be read by users who do not have the encryption key, and the data cannot be read directly from disk.

pgcrypto is installed by default when you install Greenplum Database. You must explicitly enable pgcrypto in each database in which you want to use the module.

pgcrypto allows PGP encryption using symmetric and asymmetric encryption. Symmetric encryption encrypts and decrypts data using the same key and is faster than asymmetric encryption. It is the preferred method in an environment where exchanging secret keys is not an issue. With asymmetric encryption, a public key is used to encrypt data and a private key is used to decrypt data. This is slower then symmetric encryption and it requires a stronger key.

Using pgcrypto always comes at the cost of performance and maintainability. It is important to use encryption only with the data that requires it. Also, keep in mind that you cannot search encrypted data by indexing the data.

Before you implement in-database encryption, consider the following PGP limitations.

- No support for signing. That also means that it is not checked whether the encryption sub-key belongs to the master key.
- No support for encryption key as master key. This practice is generally discouraged, so this limitation should not be a problem.
- No support for several subkeys. This may seem like a problem, as this is common practice. On the other hand, you should not use your regular GPG/PGP keys with pgcrypto, but create new ones, as the usage scenario is rather different.

Greenplum Database is compiled with zlib by default; this allows PGP encryption functions to compress data before encrypting. When compiled with OpenSSL, more algorithms will be available.

Because pgcrypto functions run inside the database server, the data and passwords move between pgcrypto and the client application in clear-text. For optimal security, you should connect locally or use SSL connections and you should trust both the system and database administrators.

pgcrypto configures itself according to the findings of the main PostgreSQL configure script.

When compiled with `zlib`, pgcrypto encryption functions are able to compress data before encrypting.

Pgcrypto has various levels of encryption ranging from basic to advanced built-in functions. The following table shows the supported encryption algorithms.

Table 61: Pgcrypto Supported Encryption Functions

Value Functionality	Built-in	With OpenSSL
MD5	yes	yes
SHA1	yes	yes
SHA224/256/384/512	yes	yes [3]
Other digest algorithms	no	yes [4]
Blowfish	yes	yes
AES	yes	yes[5]
DES/3DES/CAST5	no	yes
Raw Encryption	yes	yes
PGP Symmetric-Key	yes	yes
PGP Public Key	yes	yes

Creating PGP Keys

To use PGP asymmetric encryption in Greenplum Database, you must first create public and private keys and install them.

This section assumes you are installing Greenplum Database on a Linux machine with the Gnu Privacy Guard (gpg) command line tool. Use the latest version of GPG to create keys. Download and install Gnu Privacy Guard (GPG) for your operating system from *https://www.gnupg.org/download/*. On the GnuPG website you will find installers for popular Linux distributions and links for Windows and Mac OS X installers.

1. As root, execute the following command and choose option 1 from the menu:

```
# gpg --gen-key
gpg (GnuPG) 2.0.14; Copyright (C) 2009 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

gpg: directory '/root/.gnupg' created
gpg: new configuration file '/root/.gnupg/gpg.conf' created
gpg: WARNING: options in '/root/.gnupg/gpg.conf' are not yet active during
 this run
gpg: keyring '/root/.gnupg/secring.gpg' created
gpg: keyring '/root/.gnupg/pubring.gpg' created
Please select what kind of key you want:
 (1) RSA and RSA (default)
 (2) DSA and Elgamal
 (3) DSA (sign only)
 (4) RSA (sign only)
Your selection? 1
```

2. Respond to the prompts and follow the instructions, as shown in this example:

```
RSA keys may be between 1024 and 4096 bits long.
```

[3] SHA2 algorithms were added to OpenSSL in version 0.9.8. For older versions, pgcrypto will use built-in code.

[4] Any digest algorithm OpenSSL supports is automatically picked up. This is not possible with ciphers, which need to be supported explicitly.

[5] AES is included in OpenSSL since version 0.9.7. For older versions, pgcrypto will use built-in code.

```
What keysize do you want? (2048) Press enter to accept default key size
Requested keysize is 2048 bits
Please specify how long the key should be valid.
 0 = key does not expire
 <n> = key expires in n days
 <n>w = key expires in n weeks
 <n>m = key expires in n months
 <n>y = key expires in n years
 Key is valid for? (0) 365
Key expires at Wed 13 Jan 2016 10:35:39 AM PST
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Real name: John Doe
Email address: jdoe@email.com
Comment:
You selected this USER-ID:
 "John Doe <jdoe@email.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.
(For this demo the passphrase is blank.)
can't connect to '/root/.gnupg/S.gpg-agent': No such file or directory
You don't want a passphrase - this is probably a *bad* idea!
I will do it anyway.  You can change your passphrase at any time,
using this program with the option "--edit-key".

We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: /root/.gnupg/trustdb.gpg: trustdb created
gpg: key 2027CC30 marked as ultimately trusted
public and secret key created and signed.

gpg:  checking the trustdbgpg:
      3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg:  depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg:  next trustdb check due at 2016-01-13
pub   2048R/2027CC30 2015-01-13 [expires: 2016-01-13]
      Key fingerprint = 7EDA 6AD0 F5E0 400F 4D45   3259 077D 725E 2027
 CC30
uid                  John Doe <jdoe@email.com>
sub   2048R/4FD2EFBB 2015-01-13 [expires: 2016-01-13]
```

3. List the PGP keys by entering the following command:

```
gpg --list-secret-keys
/root/.gnupg/secring.gpg
------------------------
sec   2048R/2027CC30 2015-01-13 [expires: 2016-01-13]
uid                  John Doe <jdoe@email.com>
ssb   2048R/4FD2EFBB 2015-01-13
```

2027CC30 is the public key and will be used to *encrypt* data in the database. 4FD2EFBB is the private (secret) key and will be used to *decrypt* data.

4. Export the keys using the following commands:

```
# gpg -a --export 4FD2EFBB > public.key
# gpg -a --export-secret-keys 2027CC30 > secret.key
```

See the *pgcrypto* documentation for more information about PGP encryption functions.

Encrypting Data in Tables using PGP

This section shows how to encrypt data inserted into a column using the PGP keys you generated.

1. Dump the contents of the `public.key` file and then copy it to the clipboard:

```
# cat public.key
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)

mQENBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
vG5rJAe8PuYDSJCJ74I6w7SOH3RiRIc7IfL6xYddV42l3ctd44bl8/i71hq2UyN2
/Hbsjii2ymg7ttw3jsWAx2gP9nssDgoy8QDy/o9nNqC8EGlig96ZFnFnE6Pwbhn+
ic8MD0lK5/GAlR6Hc0ZIHf8KEcavruQlikjnABEBAAG0HHRlc3Qga2V5IDx0ZXN0
a2V5QGVtYWlsLmNvbT6JAT4EEwECACgFAlS1Zf0CGwMFCQHhM4AGCwkIBwMCBhUI
AgkKCwQWAgMBAh4BAheAAAoJEAd9cl4gJ8wwbfwH/3VyVsPkQl1owRJNxvXGt1bY
7BfrvU52yk+PPZYoes9UpdL3CMRk8gAM9bx5Sk08q2UXSZLC6fFOpEW4uWgmGYf8
JRoC3ooezTkmCBW8I1bU0qGetzVxopdXLuPGCE7hVWQe9HcSntiTLxGov1mJAwO7
TAoccXLbyuZh9Rf5vLoQdKzcCyOHh5IqXaQOT100TeFeEpb9TIiwcntg3WCSU5P0
DGoUAOanjDZ3KE8Qp7V74fhG1EZVzHb8FajR62CXSHFKqpBgiNxnTOk45NbXADn4
eTUXPSnwPi46qoAp9UQogsfGyB1XDOTB2UOqhutAMECaM7VtpePv79i0Z/NfnBe5
AQ0EVLVl/QEIANabFdQ+8QMCADOipM1bF/JrQt3zUoc4BTqICaxdyzAfz0tUSf/7
Zro2us99GlARqLWd8EqJcl/xmfcJiZyUam6ZAzzFXCgnH5Y1sdtMTJZdLp5WeOjw
gCWG/ZLu4wzxOFFzDkiPv9RDw6e5MNLtJrSp4hS5o2apKdbO4Ex83O4mJYnav/rE
iDDCWU4T0lhv3hSKCpke6LcwsX+7liozp+aNmP0Ypwfi4hR3UUMP70+V1beFqW2J
bVLz3lLLouHRgpCzla+PzzbEKs16jq77vG9kqZTCIzXoWaLljuitRlfJkO3vQ9hO
v/8yAnkcAmowZrIBlyFg2KBzhunYmN2YvkUAEQEAAYkBJQQYAQIADwUCVLVl/QIb
DAUJAeEzgAAKCRAHfXJeICfMMOHYCACFhInZA9uAM3TC44l+MrgMUJ3rW9izrO48
WrdTsxR8WkSNbIxJoWnYxYuLyPb/shc9k65huw2SSDkj//0fRrI61FPHQNPSvz62
WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8Hkk8qb5x/
HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlGWE8pvgEx
/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojDyC65KJci
Pv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=XZ8J
-----END PGP PUBLIC KEY BLOCK-----
```

2. Create a table called `userssn` and insert some sensitive data, social security numbers for Bob and Alice, in this example. Paste the public.key contents after "dearmor(".

```
CREATE TABLE userssn( ssn_id SERIAL PRIMARY KEY,
    username varchar(100), ssn bytea);

INSERT INTO userssn(username, ssn)
SELECT robotccs.username, pgp_pub_encrypt(robotccs.ssn, keys.pubkey) AS
 ssn
FROM (
        VALUES ('Alice', '123-45-6788'), ('Bob', '123-45-6799'))
            AS robotccs(username, ssn)
CROSS JOIN  (SELECT  dearmor('-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)

mQENBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
```

```
vG5rJAe8PuYDSJCJ74I6w7SOH3RiRIc7IfL6xYddV42l3ctd44bl8/i71hq2UyN2
/Hbsjii2ymg7ttw3jsWAx2gP9nssDgoy8QDy/o9nNqC8EGlig96ZFnFnE6Pwbhn+
ic8MD0lK5/GAlR6Hc0ZIHf8KEcavruQlikjnABEBAAG0HHRlc3Qga2V5IDx0ZXN0
a2V5QGVtYWlsLmNvbT6JAT4EEwECACgFAlS1Zf0CGwMFCQHhM4AGCwkIBwMCBhUI
AgkKCwQWAgMBAh4BAheAAAoJEAd9cl4gJ8wwbfwH/3VyVsPkQl1owRJNxvXGt1bY
7BfrvU52yk+PPZYoes9UpdL3CMRk8gAM9bx5Sk08q2UXSZLC6fFOpEW4uWgmGYf8
JRoC3ooezTkmCBW8I1bU0qGetzVxopdXLuPGCE7hVWQe9HcSntiTLxGov1mJAwO7
TAoccXLbyuZh9Rf5vLoQdKzcCyOHh5IqXaQOT100TeFeEpb9TIiwcntg3WCSU5P0
DGoUAOanjDZ3KE8Qp7V74fhG1EZVzHb8FajR62CXSHFKqpBgiNxnTOk45NbXADn4
eTUXPSnwPi46qoAp9UQogsfGyB1XDOTB2UOqhutAMECaM7VtpePv79i0Z/NfnBe5
AQ0EVLVl/QEIANabFdQ+8QMCADOipM1bF/JrQt3zUoc4BTqICaxdyzAfz0tUSf/7
Zro2us99GlARqLWd8EqJcl/xmfcJiZyUam6ZAzzFXCgnH5Y1sdtMTJZdLp5WeOjw
gCWG/ZLu4wzxOFFzDkiPv9RDw6e5MNLtJrSp4hS5o2apKdbO4Ex83O4mJYnav/rE
iDDCWU4T0lhv3hSKCpke6LcwsX+7liozp+aNmP0Ypwfi4hR3UUMP70+V1beFqW2J
bVLz3lLLouHRgpCzla+PzzbEKs16jq77vG9kqZTCIzXoWaLljuitRlfJkO3vQ9hO
v/8yAnkcAmowZrIBlyFg2KBzhunYmN2YvkUAEQEAAYkBJQQYAQIADwUCVLVl/QIb
DAUJAeEzgAAKCRAHfXJeICfMMOHYCACFhInZA9uAM3TC44l+MrgMUJ3rW9izrO48
WrdTsxR8WkSNbIxJoWnYxYuLyPb/shc9k65huw2SSDkj//0fRrI61FPHQNPSvz62
WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8Hkk8qb5x/
HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlGWE8pvgEx
/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojDyC65KJci
Pv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=XZ8J
-----END PGP PUBLIC KEY BLOCK-----' AS pubkey) AS keys;
```

3. Verify that the `ssn` column is encrypted.

```
test_db=# select * from userssn;
ssn_id   | 1
username | Alice
ssn      | \301\300L\003\235M%_O
\322\357\273\001\010\000\272\227\010\341\216\360\217C\020\261)_\367
[\227\034\313:C\354d<\337\006Q\351('\2330\031lX\263Qf
\341\262\200\3015\235\036AK\242fL+\315g\322
7u\270*\304\361\355\220\021\330"\200%\264\274}R
\213\377\363\235\366\030\023)\364!\331\303\237t\277=
f \015\004\242\231\263\225%\032\271a\001\035\277\021\375X\232\304\305/
\340\334\0131\325\344[~\362\0
37-\251\336\303\340\377_\011\275\301/MY\334\343\245\244\372y\257S
\374\230\346\277\373W\346\230\276\
017fi\226Q\307\012\326\3646\000\326\005:E\364W\252=zz\010(:\343Y
\237\257iqU\0326\350=v0\362\327\350\
315G^\027:K_9\254\362\354\215<\001\304\357\331\355\323,\302\213Fe
\265\315\232\367\254\245%(\\\373
4\254\230\331\356\006B\257\333\326H\022\013\353\216F?\023\220\370\035vH5/
\227\344b\322\227\026\362=\
42\033\322<\001}\243\224;)\030zqX\214\340\221\035\275U
\345\327\214\032\351\223c\2442\345\304K\016\
011\214\307\227\237\270\026'R\205\205a~1\263\236[\037C
\260\031\205\374\245\317\033k|\366\253\037
---------
+----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------
ssn_id   | 2
username | Bob
```

```
ssn      | \301\300L\003\235M%_O\322\357\273\001\007\377t>\345\343,
\200\256\272\300\012\033M4\265\032L
L[v\262k\244\2435\264\232B\357\370d9\375\011\002\327\235<\246\210b
\030\012\337@\226Z\361\246\032\00
7'\012c\353]\355d7\360T\335\314\367\370;X\371\350*\231\212\260B
\010#RQ0\223\253c7\0132b\355\242\233\34
1\000\370\370\366\013\022\357\005i\202~\005\\z\301o\012\230Z
\014\362\244\324&\243g\351\362\325\375
\213\032\226$\2751\256XR\346k\266\030\234\267\201vUh\004\250\337A\231\223u
\247\366/i\022\275\276\350\2
20\316\306|\203+\010\261;\232\254tp\255\243\261\373Rq;\316w
\357\006\207\374U\333\365\365\245hg\031\005
\322\347ea\220\015l\212g\337\264\336b\263\004\311\210.4\340G+\221\274D
\035\375\2216\241'\346a0\273wE\2
12\342y^\202\262|A7\202t\240\333p\345G\373\253\243oCO
\011\360\247\211\014\024{\272\271\322<\001\267
\347\240\005\213\0078\036\210\307$\317\322\311\222\035\354\006<
\266\264\004\376\251q\256\220(+\030\
3270\013c\327\272\212%\363\033\252\322\337\354\276\225\232\201\212^
\304\210\2269@\3230\370{
```

4. Extract the public.key ID from the database:

```
SELECT pgp_key_id(dearmor('-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)

mQENBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
vG5rJAe8PuYDSJCJ74I6w7SOH3RiRIc7IfL6xYddV42l3ctd44bl8/i71hq2UyN2
/Hbsjii2ymg7ttw3jsWAx2gP9nssDgoy8QDy/o9nNqC8EGlig96ZFnFnE6Pwbhn+
ic8MD0lK5/GAlR6Hc0ZIHf8KEcavruQlikjnABEBAAG0HHRlc3Qga2V5IDx0ZXN0
a2V5QGVtYWlsLmNvbT6JAT4EEwECACgFAlS1Zf0CGwMFCQHhM4AGCwkIBwMCBhUI
AgkKCwQWAgMBAh4BAheAAAoJEAd9cl4gJ8wwbfwH/3VyVsPkQl1owRJNxvXGt1bY
7BfrvU52yk+PPZYoes9UpdL3CMRk8gAM9bx5Sk08q2UXSZLC6fFOpEW4uWgmGYf8
JRoC3ooezTkmCBW8I1bU0qGetzVxopdXLuPGCE7hVWQe9HcSntiTLxGov1mJAwO7
TAoccXLbyuZh9Rf5vLoQdKzcCyOHh5IqXaQOT100TeFeEpb9TIiwcntg3WCSU5P0
DGoUAOanjDZ3KE8Qp7V74fhG1EZVzHb8FajR62CXSHFKqpBgiNxnTOk45NbXADn4
eTUXPSnwPi46qoAp9UQogsfGyB1XDOTB2UOqhutAMECaM7VtpePv79i0Z/NfnBe5
AQ0EVLVl/QEIANabFdQ+8QMCADOipM1bF/JrQt3zUoc4BTqICaxdyzAfz0tUSf/7
Zro2us99GlARqLWd8EqJcl/xmfcJiZyUam6ZAzzFXCgnH5Y1sdtMTJZdLp5WeOjw
gCWG/ZLu4wzxOFFzDkiPv9RDw6e5MNLtJrSp4hS5o2apKdbO4Ex83O4mJYnav/rE
iDDCWU4T0lhv3hSKCpke6LcwsX+7liozp+aNmP0Ypwfi4hR3UUMP70+V1beFqW2J
bVLz3lLLouHRgpCzla+PzzbEKs16jq77vG9kqZTCIzXoWaLljuitRlfJkO3vQ9hO
v/8yAnkcAmowZrIBlyFg2KBzhunYmN2YvkUAEQEAAYkBJQQYAQIADwUCVLVl/QIb
DAUJAeEzgAAKCRAHfXJeICfMMOHYCACFhInZA9uAM3TC44l+MrgMUJ3rW9izrO48
WrdTsxR8WkSNbIxJoWnYxYuLyPb/shc9k65huw2SSDkj//0fRrI61FPHQNPSvz62
WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8Hkk8qb5x/
HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlGWE8pvgEx
/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojDyC65KJci
Pv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=XZ8J
-----END PGP PUBLIC KEY BLOCK-----'));

pgp_key_id | 9D4D255F4FD2EFBB
```

This shows that the PGP key ID used to encrypt the `ssn` column is 9D4D255F4FD2EFBB. It is recommended to perform this step whenever a new key is created and then store the ID for tracking.

You can use this key to see which key pair was used to encrypt the data:

```
SELECT username, pgp_key_id(ssn) As key_used
FROM userssn;
```

```
username | Bob
key_used | 9D4D255F4FD2EFBB
---------+-----------------
username | Alice
key_used | 9D4D255F4FD2EFBB
```

Note: Different keys may have the same ID. This is rare, but is a normal event. The client application should try to decrypt with each one to see which fits — like handling ANYKEY. See *pgp_key_id()* in the pgcrypto documentation.

5. Decrypt the data using the private key.

```
SELECT username, pgp_pub_decrypt(ssn, keys.privkey)
                 AS decrypted_ssn FROM userssn
                 CROSS JOIN
                 (SELECT dearmor('-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)

lQOYBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
vG5rJAe8PuYDSJCJ74I6w7SOH3RiRIc7IfL6xYddV42l3ctd44bl8/i71hq2UyN2
/Hbsjii2ymg7ttw3jsWAx2gP9nssDgoy8QDy/o9nNqC8EGlig96ZFnFnE6Pwbhn+
ic8MD0lK5/GAlR6Hc0ZIHf8KEcavruQlikjnABEBAAEAB/wNfjjvP1brRfjjIm/j
XwUNm+sI4v2Ur7qZC94VTukPGf67lvqcYZJuqXxvZrZ8bl6mvl65xEUiZYy7BNA8
fe0PaM4Wy+Xr94Cz2bPbWgawnRNN3GAQy4rlBTrvqQWy+kmpbd87iTjwZidZNNmx
02iSzraq41Rt0Zx21Jh4rkpF67ftmzOH0vlrS0bWOvHUeMY7tCwmdPe9HbQeDlPr
n9CllUqBn4/acTtCClWAjREZn0zXAsNixtTIPC1V+9nO9YmecMkVwNfIPkIhymAM
OPFnuZ/Dz1rCRHjNHb5j6ZyUM5zDqUVnnezktxqrOENSxm0gfMGcpxHQogUMzb7c
6UyBBADSCXHPfo/VPVtMm5p1yGrNOR2jR2rUj9+poZzD2gjkt5G/xIKRlkB4uoQl
emu27wr9dVEX7ms0nvDq58iutbQ4d0JIDlcHMeSRQZluErblB75Vj3HtImblPjpn
4Jx6SWRXPUJPGXGI87u0UoBH0Lwij7M2PW7l1ao+MLEA9jAjQwQA+sr9BKPL4Ya2
r5nE72gsbCCLowkC0rdldf1RGtobwYDMpmYZhOaRKjkOTMG6rCXJxrf6LqiN8w/L
/gNziTmch35MCq/MZzA/bN4VMPyeIlwzxVZkJLsQ7yyqX/A7ac7B7DH0KfXciEXW
MSOAJhMmklW1Q1RRNw3cnYi8w3q7X40EAL/w54FVvvPqp3+sCd86SAAapM4UO2R3
tIsuNVemMWdgNXwvK8AJsz7VreVU5yZ4B8hvCuQj1C7geaN/LXhiT8foRsJC5o71
Bf+iHC/VNEv4k4uDb4lOgnHJYYyifB1wC+nn/EnXCZYQINMia1a4M6Vqc/RIfTH4
nwkZt/89LsAiR/20HHRlc3Qga2V5IDx0ZXN0a2V5QGVtYWlsLmNvbT6JAT4EEwEC
ACgFAlS1Zf0CGwMFCQHhM4AGCwkIBwMCBhUIAgkKCwQWAgMBAh4BAheAAAoJEAd9
cl4gJ8wwbfwH/3VyVsPkQl1owRJNxvXGt1bY7BfrvU52yk+PPZYoes9UpdL3CMRk
8gAM9bx5Sk08q2UXSZLC6fFOpEW4uWgmGYf8JRoC3ooezTkmCBW8I1bU0qGetzVx
opdXLuPGCE7hVWQe9HcSntiTLxGov1mJAwO7TAoccXLbyuZh9Rf5vLoQdKzcCyOH
h5IqXaQOT100TeFeEpb9TIiwcntg3WCSU5P0DGoUAOanjDZ3KE8Qp7V74fhG1EZV
zHb8FajR62CXSHFKqpBgiNxnTOk45NbXADn4eTUXPSnwPi46qoAp9UQogsfGyB1X
DOTB2UOqhutAMECaM7VtpePv79i0Z/NfnBedA5gEVLVl/QEIANabFdQ+8QMCADOi
pM1bF/JrQt3zUoc4BTqICaxdyzAfz0tUSf/7Zro2us99GlARqLWd8EqJcl/xmfcJ
iZyUam6ZAzzFXCgnH5Y1sdtMTJZdLp5WeOjwgCWG/ZLu4wzxOFFzDkiPv9RDw6e5
MNLtJrSp4hS5o2apKdbO4Ex83O4mJYnav/rEiDDCWU4T0lhv3hSKCpke6LcwsX+7
liozp+aNmP0Ypwfi4hR3UUMP70+V1beFqW2JbVLz3lLLouHRgpCzla+PzzbEKs16
jq77vG9kqZTCIzXoWaLljuitRlfJkO3vQ9hOv/8yAnkcAmowZrIBlyFg2KBzhunY
mN2YvkUAEQEAAQAH/A7r4hDrnmzX3QU6FAzePlRB7niJtE2IEN8AufF05Q2PzKU/
c1S72WjtqMAIAgYasDkOhfhcxanTneGuFVYggKT3eSDm1RFKpRjX22m0zKdwy67B
Mu95V2Oklul6OCm8dO6+2fmkGxGqc4ZsKy+jQxtxK3HG9YxMC0dvA2v2C5N4TWi3
Utc7zh//k6IbmaLd7F1d7DXt7Hn2Qsmo8I1rtgPE8grDToomTnRUodToyejEqKyI
ORwsp8n8g2CSFaXSrEyU6HbFYXSxZealhQJGYLFOZdR0MzVtZQCn/7n+IHjupndC
Nd2a8DVx3yQS3dAmvLzhFacZdjXi31wvj0moFOkEAOCz1E63SKNNksniQ11lRMJp
gaov6Ux/zGLMstwTzNouI+Kr8/db0GlSAy1Z3UoAB4tFQXEApoX9A4AJ2KqQjqOX
cZVULenfDZaxrbb9Lid7ZnTDXKVyGTWDF7ZHavHJ4981mCW17lU11zHBB9xMlx6p
dhFvb0gdy0jSLaFMFr/JBAD0fz3RrhP7e6Xll2zdBqGthjC5S/IoKwwBgw6ri2yx
LoxqBr2pl9PotJJ/JUMPhD/LxuTcOZtYjy8PKgm5jhnBDq3Ss0kNKAY1f5EkZG9a
6I4iAX/NekqSyF+OgBfC9aCgS5RG8hYoOCbp8na5R3bgiuS8IzmVmm5OhZ4MDEwg
nQP7BzmR0p5BahpZ8r3Ada7FcK+0ZLLRdLmOYF/yUrZ53SoYCZRzU/GmtQ7LkXBh
Gjqied9Bs1MHdNUolq7GaexcjZmOWHEf6w9+9M4+vxtQq1nkIWqtaphewEmd5/nf
EP3sIY0EAE3mmiLmHLqBju+UJKMNwFNeyMTqgcg50ISH8J9FRIkBJQQYAQIADwUC
```

```
VLVl/QIbDAUJAeEzgAAKCRAHfXJeICfMMOHYCACFhInZA9uAM3TC44l+MrgMUJ3r
W9izrO48WrdTsxR8WkSNbIxJoWnYxYuLyPb/shc9k65huw2SSDkj//0fRrI61FPH
QNPSvz62WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8H
kk8qb5x/HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlG
WE8pvgEx/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojD
yC65KJciPv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=fa+6
-----END PGP PRIVATE KEY BLOCK-----') AS privkey) AS keys;

username | decrypted_ssn
----------+---------------
 Alice    | 123-45-6788
 Bob      | 123-45-6799
(2 rows)
```

If you created a key with passphrase, you may have to enter it here. However for the purpose of this example, the passphrase is blank.

Key Management

Whether you are using symmetric (single private key) or asymmetric (public and private key) cryptography, it is important to store the master or private key securely. There are many options for storing encryption keys, for example, on a file system, key vault, encrypted USB, trusted platform module (TPM), or hardware security module (HSM).

Consider the following questions when planning for key management:

- Where will the keys be stored?
- When should keys expire?
- How are keys protected?
- How are keys accessed?
- How can keys be recovered and revoked?

The Open Web Application Security Project (OWASP) provides a very comprehensive *guide to securing encryption keys*.

Security Best Practices

Describes basic security best practices that you should follow to ensure the highest level of system security.

In the default Greenplum Database security configuration:

- Only local connections are allowed.
- Basic authentication is configured for the superuser (`gpadmin`).
- The superuser is authorized to do anything.
- Only database role passwords are encrypted.

System User (gpadmin)

Secure and limit access to the `gpadmin` system user.

Greenplum requires a UNIX user id to install and initialize the Greenplum Database system. This system user is referred to as `gpadmin` in the Greenplum documentation. The `gpadmin` user is the default database superuser in Greenplum Database, as well as the file system owner of the Greenplum installation and its underlying data files. The default administrator account is fundamental to the design of Greenplum Database. The system cannot run without it, and there is no way to limit the access of the `gpadmin` user id.

The `gpadmin` user can bypass all security features of Greenplum Database. Anyone who logs on to a Greenplum host with this user id can read, alter, or delete any data, including system catalog data and database access rights. Therefore, it is very important to secure the `gpadmin` user id and only allow essential system administrators access to it.

Administrators should only log in to Greenplum as `gpadmin` when performing certain system maintenance tasks (such as upgrade or expansion).

Database users should never log on as `gpadmin`, and ETL or production workloads should never run as `gpadmin`.

Superusers

Roles granted the `SUPERUSER` attribute are superusers. Superusers bypass all access privilege checks and resource queues. Only system administrators should be given superuser rights.

See "Altering Role Attributes" in the *Greenplum Database Administrator Guide*.

Login Users

Assign a distinct role to each user who logs in and set the `LOGIN` attribute.

For logging and auditing purposes, each user who is allowed to log in to Greenplum Database should be given their own database role. For applications or web services, consider creating a distinct role for each application or service. See "Creating New Roles (Users)" in the *Greenplum Database Administrator Guide*.

Each login role should be assigned to a single, non-default resource queue.

Groups

Use groups to manage access privileges.

Create a group for each logical grouping of object/access permissions.

Every login user should belong to one or more roles. Use the `GRANT` statement to add group access to a role. Use the `REVOKE` statement to remove group access from a role.

The `LOGIN` attribute should not be set for group roles.

See "Creating Groups (Role Membership)" in the *Greenplum Database Administrator Guide*.

Object Privileges

Only the owner and superusers have full permissions to new objects. Permission must be granted to allow other rules (users or groups) to access objects. Each type of database object has different privileges that may be granted. Use the `GRANT` statement to add a permission to a role and the `REVOKE` statement to remove the permission.

You can change the owner of an object using the `REASIGN OWNED BY` statement. For example, to prepare to drop a role, change the owner of the objects that belong to the role. Use the `DROP OWNED BY` to drop objects, including dependent objects, that are owned by a role.

Schemas can be used to enforce an additional layer of object permissions checking, but schema permissions do not override object privileges set on objects contained within the schema.

Operating System Users and File System

To protect the network from intrusion, system administrators should verify the passwords used within an organization are sufficiently strong. The following recommendations can strengthen a password:

- Minimum password length recommendation: At least 9 characters. MD5 passwords should be 15 characters or longer.
- Mix upper and lower case letters.
- Mix letters and numbers.
- Include non-alphanumeric characters.
- Pick a password you can remember.

The following are recommendations for password cracker software that you can use to determine the strength of a password.

- John The Ripper. A fast and flexible password cracking program. It allows the use of multiple word lists and is capable of brute-force password cracking. It is available online at *http://www.openwall.com/john/*.
- Crack. Perhaps the most well-known password cracking software, Crack is also very fast, though not as easy to use as John The Ripper. It can be found online at *https://dropsafe.crypticide.com/alecm/software/crack/c50-faq.html*.

The security of the entire system depends on the strength of the root password. This password should be at least 12 characters long and include a mix of capitalized letters, lowercase letters, special characters, and numbers. It should not be based on any dictionary word.

Password expiration parameters should be configured.

Ensure the following line exists within the file `/etc/libuser.conf` under the `[import]` section.

```
login_defs = /etc/login.defs
```

Ensure no lines in the `[userdefaults]` section begin with the following text, as these words override settings from `/etc/login.defs`:

- `LU_SHADOWMAX`
- `LU_SHADOWMIN`
- `LU_SHADOWWARNING`

Ensure the following command produces no output. Any accounts listed by running this command should be locked.

```
grep "^+:" /etc/passwd /etc/shadow /etc/group
```

Note: We strongly recommend that customers change their passwords after initial setup.

```
cd /etc
chown root:root passwd shadow group gshadow
chmod 644 passwd group
chmod 400 shadow gshadow
```

Find all the files that are world-writable and that do not have their sticky bits set.

```
find / -xdev -type d \( -perm -0002 -a ! -perm -1000 \) -print
```

Set the sticky bit (`# chmod +t {dir}`) for all the directories that result from running the previous command.

Find all the files that are world-writable and fix each file listed.

```
find / -xdev -type f -perm -0002 -print
```

Set the right permissions (`# chmod o-w {file}`) for all the files generated by running the aforementioned command.

Find all the files that do not belong to a valid user or group and either assign an owner or remove the file, as appropriate.

```
find / -xdev \( -nouser -o -nogroup \) -print
```

Find all the directories that are world-writable and ensure they are owned by either root or a system account (assuming only system accounts have a User ID lower than 500). If the command generates any output, verify the assignment is correct or reassign it to root.

```
find / -xdev -type d -perm -0002 -uid +500 -print
```

Authentication settings such as password quality, password expiration policy, password reuse, password retry attempts, and more can be configured using the Pluggable Authentication Modules (PAM) framework. PAM looks in the directory `/etc/pam.d` for application-specific configuration information. Running `authconfig` or `system-config-authentication` will re-write the PAM configuration files, destroying any manually made changes and replacing them with system defaults.

The default `pam_cracklib` PAM module provides strength checking for passwords. To configure `pam_cracklib` to require at least one uppercase character, lowercase character, digit, and special character, as recommended by the U.S. Department of Defense guidelines, edit the file `/etc/pam.d/system-auth` to include the following parameters in the line corresponding to password requisite `pam_cracklib.so try_first_pass`.

```
retry=3:
dcredit=-1. Require at least one digit
ucredit=-1. Require at least one upper case character
ocredit=-1. Require at least one special character
lcredit=-1. Require at least one lower case character
minlen-14. Require a minimum password length of 14.
```

For example:

```
```

```
password required pam_cracklib.so try_first_pass retry=3\minlen=14
 dcredit=-1 ucredit=-1 ocredit=-1 lcredit=-1
```

These parameters can be set to reflect your security policy requirements. Note that the password restrictions are not applicable to the root password.

The `pam_tally2` PAM module provides the capability to lock out user accounts after a specified number of failed login attempts. To enforce password lockout, edit the file `/etc/pam.d/system-auth` to include the following lines:

- The first of the auth lines should include:

  ```
  auth required pam_tally2.so deny=5 onerr=fail unlock_time=900
  ```

- The first of the account lines should include:

  ```
  account required pam_tally2.so
  ```

Here, the deny parameter is set to limit the number of retries to 5 and the `unlock_time` has been set to 900 seconds to keep the account locked for 900 seconds before it is unlocked. These parameters may be configured appropriately to reflect your security policy requirements. A locked account can be manually unlocked using the `pam_tally2` utility:

```

/sbin/pam_tally2 --user {username} -reset
```

You can use PAM to limit the reuse of recent passwords. The remember option for the `pam_ unix` module can be set to remember the recent passwords and prevent their reuse. To accomplish this, edit the appropriate line in `/etc/pam.d/system-auth` to include the remember option.

For example:

```

password sufficient pam_unix.so [ … existing_options …]
remember=5
```

You can set the number of previous passwords to remember to appropriately reflect your security policy requirements.

```

cd /etc
chown root:root passwd shadow group gshadow
chmod 644 passwd group
chmod 400 shadow gshadow
```

Chapter 5

Greenplum Database Best Practices

A best practice is a method or technique that has consistently shown results superior to those achieved with other means. Best practices are found through experience and are proven to reliably lead to a desired result. Best practices are a commitment to use any product correctly and optimally, by leveraging all the knowledge and expertise available to ensure success.

This document does not teach you how to use Greenplum Database features. Links are provided to other relevant parts of the Greenplum Database documentation for information on how to use and implement specific Greenplum Database features. This document addresses the most important best practices to follow when designing, implementing, and using Greenplum Database.

It is not the intent of this document to cover the entire product or compendium of features, but rather to provide a summary of *what matters most* in Greenplum Database. This document does not address *edge* use cases. While edge use cases can further leverage and benefit from Greenplum Database features, they require a proficient knowledge and expertise with these features, as well as a deep understanding of your environment, including SQL access, query execution, concurrency, workload, and other factors.

By mastering these best practices, you will increase the success of your Greenplum Database clusters in the areas of maintenance, support, performance, and scalability.

Best Practices Summary

A summary of best practices for Greenplum Database.

Data Model

Greenplum Database is an analytical MPP shared-nothing database. This model is significantly different from a highly normalized/transactional SMP database. Because of this, the following best practices are recommended.

- Greenplum Database performs best with a denormalized schema design suited for MPP analytical processing for example, Star or Snowflake schema, with large fact tables and smaller dimension tables.
- Use the same data types for columns used in joins between tables.

See *Schema Design*.

Heap vs. Append-Optimized Storage

- Use heap storage for tables and partitions that will receive iterative batch and singleton `UPDATE`, `DELETE`, and `INSERT` operations.
- Use heap storage for tables and partitions that will receive concurrent `UPDATE`, `DELETE`, and `INSERT` operations.
- Use append-optimized storage for tables and partitions that are updated infrequently after the initial load and have subsequent inserts only performed in large batch operations.
- Avoid performing singleton `INSERT`, `UPDATE`, or `DELETE` operations on append-optimized tables.
- Avoid performing concurrent batch `UPDATE` or `DELETE` operations on append-optimized tables. Concurrent batch `INSERT` operations are acceptable.

See *Heap Storage or Append-Optimized Storage*.

Row vs. Column Oriented Storage

- Use row-oriented storage for workloads with iterative transactions where updates are required and frequent inserts are performed.
- Use row-oriented storage when selects against the table are wide.
- Use row-oriented storage for general purpose or mixed workloads.
- Use column-oriented storage where selects are narrow and aggregations of data are computed over a small number of columns.
- Use column-oriented storage for tables that have single columns that are regularly updated without modifying other columns in the row.

See *Row or Column Orientation*.

Compression

- Use compression on large append-optimized and partitioned tables to improve I/O across the system.
- Set the column compression settings at the level where the data resides.
- Balance higher levels of compression with the time and CPU cycles needed to compress and uncompress data.

See *Compression*.

Distributions

- Explicitly define a column or random distribution for all tables. Do not use the default.
- Use a single column that will distribute data across all segments evenly.

- Do not distribute on columns that will be used in the `WHERE` clause of a query.
- Do not distribute on dates or timestamps.
- Never distribute and partition tables on the same column.
- Achieve local joins to significantly improve performance by distributing on the same column for large tables commonly joined together.
- To ensure there is no data skew, validate that data is evenly distributed after the initial load and after incremental loads.

See *Distributions*.

Resource Queue Memory Management

- Set `vm.overcommit_memory` to 2.
- Do not configure the OS to use huge pages.
- Use `gp_vmem_protect_limit` to set the maximum memory that the instance can allocate for *all* work being done in each segment database.
- You can use `gp_vmem_protect_limit` by calculating:
  - `gp_vmem` – the total memory available to Greenplum Database

    ```
    gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM)) / 1.7
    ```

    where `SWAP` is the host's swap space in GB, and `RAM` is the host's RAM in GB
  - `max_acting_primary_segments` – the maximum number of primary segments that could be running on a host when mirror segments are activated due to a host or segment failure
  - `gp_vmem_protect_limit`

    ```
    gp_vmem_protect_limit = gp_vmem / acting_primary_segments
    ```

    Convert to MB to set the value of the configuration parameter.
- In a scenario where a large number of workfiles are generated calculate the `gp_vmem` factor with this formulat to account for the workfiles:

  ```
  gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM - (300KB *
        total_#_workfiles))) / 1.7
  ```
- Never set `gp_vmem_protect_limit` too high or larger than the physical RAM on the system.
- Use the calculated `gp_vmem` value to calculate the setting for the `vm.overcommit_ratio` operating system parameter:

  ```
  vm.overcommit_ratio = (RAM - 0.026 * gp_vmem) / RAM
  ```
- Use `statement_mem` to allocate memory used for a query per segment db.
- Use resource queues to set both the numbers of active queries (`ACTIVE_STATEMENTS`) and the amount of memory (`MEMORY_LIMIT`) that can be utilized by queries in the queue.
- Associate all users with a resource queue. Do not use the default queue.
- Set `PRIORITY` to match the real needs of the queue for the workload and time of day. Avoid using MAX priority.
- Ensure that resource queue memory allocations do not exceed the setting for `gp_vmem_protect_limit`.
- Dynamically update resource queue settings to match daily operations flow.

See *Setting the Greenplum Recommended OS Parameters* and *Memory and Resource Management with Resource Queues*.

Partitioning

- Partition large tables only. Do not partition small tables.
- Use partitioning only if partition elimination (partition pruning) can be achieved based on the query criteria.
- Choose range partitioning over list partitioning.
- Partition the table based on a commonly-used column, such as a date column.
- Never partition and distribute tables on the same column.
- Do not use default partitions.
- Do not use multi-level partitioning; create fewer partitions with more data in each partition.
- Validate that queries are selectively scanning partitioned tables (partitions are being eliminated) by examining the query `EXPLAIN` plan.
- Do not create too many partitions with column-oriented storage because of the total number of physical files on every segment: `physical files = segments x columns x partitions`

See *Partitioning*.

Indexes

- In general indexes are not needed in Greenplum Database.
- Create an index on a single column of a columnar table for drill-through purposes for high cardinality tables that require queries with high selectivity.
- Do not index columns that are frequently updated.
- Consider dropping indexes before loading data into a table. After the load, re-create the indexes for the table.
- Create selective B-tree indexes.
- Do not create bitmap indexes on columns that are updated.
- Avoid using bitmap indexes for unique columns, very high or very low cardinality data. Bitmap indexes perform best when the column has a low cardinality—100 to 100,000 distinct values.
- Do not use bitmap indexes for transactional workloads.
- In general do not index partitioned tables. If indexes are needed, the index columns must be different than the partition columns.

See *Indexes*.

Resource Queues

- Use resource queues to manage the workload on the cluster.
- Associate all roles with a user-defined resource queue.
- Use the `ACTIVE_STATEMENTS` parameter to limit the number of active queries that members of the particular queue can run concurrently.
- Use the `MEMORY_LIMIT` parameter to control the total amount of memory that queries running through the queue can utilize.
- Alter resource queues dynamically to match the workload and time of day.

See *Configuring Resource Queues*.

Monitoring and Maintenance

- Implement the "Recommended Monitoring and Maintenance Tasks" in the *Greenplum Database Administrator Guide*.
- Run `gpcheckperf` at install time and periodically thereafter, saving the output to compare system performance over time.
- Use all the tools at your disposal to understand how your system behaves under different loads.
- Examine any unusual event to determine the cause.

- Monitor query activity on the system by running explain plans periodically to ensure the queries are running optimally.
- Review plans to determine whether index are being used and partition elimination is occurring as expected.
- Know the location and content of system log files and monitor them on a regular basis, not just when problems arise.

See *System Monitoring and Maintenance*, *Query Profiling* and *Monitoring Greenplum Database Log Files*.

ANALYZE

- Determine if analyzing the database is actually needed. Analyzing is not needed if `gp_autostats_mode` is set to `on_no_stats` (the default) and the table is not partitioned.
- Use `analyzedb` in preference to `ANALYZE` when dealing with large sets of tables, as it does not require analyzing the entire database. The `analyzedb` utility updates statistics data for the specified tables incrementally and concurrently. For append optimized tables, `analyzedb` updates statistics incrementally only if the statistics are not current. For heap tables, statistics are always updated. `ANALYZE` does not update the table metadata that the `analyzedb` utility uses to determine whether table statistics are up to date.
- Selectively run `ANALYZE` at the table level when needed.
- Always run `ANALYZE` after `INSERT`, `UPDATE`. and `DELETE` operations that significantly changes the underlying data.
- Always run `ANALYZE` after `CREATE INDEX` operations.
- If `ANALYZE` on very large tables takes too long, run `ANALYZE` only on the columns used in a join condition, `WHERE` clause, `SORT`, `GROUP BY`, or `HAVING` clause.
- When dealing with large sets of tables, use `analyzedb` instead of `ANALYZE`.

See *Updating Statistics with ANALYZE*.

Vacuum

- Run `VACUUM` after large `UPDATE` and `DELETE` operations.
- Do not run `VACUUM FULL`. Instead run a `CREATE TABLE...AS` operation, then rename and drop the original table.
- Frequently run `VACUUM` on the system catalogs to avoid catalog bloat and the need to run `VACUUM FULL` on catalog tables.
- Never kill `VACUUM` on catalog tables.

See *Managing Bloat in a Database*.

Loading

- Maximize the parallelism as the number of segments increase.
- Spread the data evenly across as many ETL nodes as possible.
    - Split very large data files into equal parts and spread the data across as many file systems as possible.
    - Run two `gpfdist` instances per file system.
    - Run `gpfdist` on as many interfaces as possible.
    - Use `gp_external_max_segs` to control the number of segments that will request data from the `gpfdist` process.
    - Always keep `gp_external_max_segs` and the number of `gpfdist` processes an even factor.
- Always drop indexes before loading into existing tables and re-create the index after loading.
- Run `VACUUM` after load errors to recover space.

See *Loading Data*.

Security

- Secure the `gpadmin` user id and only allow essential system administrators access to it.
- Administrators should only log in to Greenplum as `gpadmin` when performing certain system maintenance tasks (such as upgrade or expansion).
- Limit users who have the SUPERUSER role attribute. Roles that are superusers bypass all access privilege checks in Greenplum Database, as well as resource queuing. Only system administrators should be given superuser rights. See "Altering Role Attributes" in the *Greenplum Database Administrator Guide*.
- Database users should never log on as `gpadmin`, and ETL or production workloads should never run as `gpadmin`.
- Assign a distinct Greenplum Database role to each user, application, or service that logs in.
- For applications or web services, consider creating a distinct role for each application or service.
- Use groups to manage access privileges.
- Protect the root password.
- Enforce a strong password password policy for operating system passwords.
- Ensure that important operating system files are protected.

See *Security*.

Encryption

- Encrypting and decrypting data has a performance cost; only encrypt data that requires encryption.
- Do performance testing before implementing any encryption solution in a production system.
- Server certificates in a production Greenplum Database system should be signed by a certificate authority (CA) so that clients can authenticate the server. The CA may be local if all clients are local to the organization.
- Client connections to Greenplum Database should use SSL encryption whenever the connection goes through an insecure link.
- A symmetric encryption scheme, where the same key is used to both encrypt and decrypt, has better performance than an asymmetric scheme and should be used when the key can be shared safely.
- Use crytographic functions to encrypt data on disk. The data is encrypted and decrypted in the database process, so it is important to secure the client connection with SSL to avoid transmitting unencrypted data.
- Use the gpfdists protocol to secure ETL data as it is loaded into or unloaded from the database. .

See *Encrypting Data and Database Connections*

High Availability

> **Note:** The following guidelines apply to actual hardware deployments, but not to public cloud-based infrastructure, where high availability solutions may already exist.

- Use a hardware RAID storage solution with 8 to 24 disks.
- Use RAID 1, 5, or 6 so that the disk array can tolerate a failed disk.
- Configure a hot spare in the disk array to allow rebuild to begin automatically when disk failure is detected.
- Protect against failure of the entire disk array and degradation during rebuilds by mirroring the RAID volume.
- Monitor disk utilization regularly and add additional space when needed.
- Monitor segment skew to ensure that data is distributed evenly and storage is consumed evenly at all segments.
- Set up a standby master instance to take over if the primary master fails.
- Plan how to switch clients to the new master instance when a failure occurs, for example, by updating the master address in DNS.

- Set up monitoring to send notifications in a system monitoring application or by email when the primary fails.
- Set up mirrors for all segments.
- Locate primary segments and their mirrors on different hosts to protect against host failure.
- Recover failed segments promptly, using the `gprecoverseg` utility, to restore redundancy and return the system to optimal balance.
- Consider a Dual Cluster configuration to provide an additional level of redundancy and additional query processing throughput.
- Backup Greenplum databases regularly unless the data is easily restored from sources.
- If backups are saved to local cluster storage, move the files to a safe, off-cluster location when the backup is complete.
- If backups are saved to NFS mounts, use a scale-out NFS solution such as Dell EMC Isilon to prevent IO bottlenecks.
- Consider using Greenplum integration to stream backups to the Dell EMC Data Domain enterprise backup platform.

See *High Availability*.

System Configuration

Requirements and best practices for system administrators who are configuring Greenplum Database cluster hosts.

Configuration of the Greenplum Database cluster is usually performed as root.

Configuring the Timezone

Greenplum Database selects a timezone to use from a set of internally stored PostgreSQL timezones. The available PostgreSQL timezones are taken from the Internet Assigned Numbers Authority (IANA) Time Zone Database, and Greenplum Database updates its list of available timezones as necessary when the IANA database changes for PostgreSQL.

Greenplum selects the timezone by matching a PostgreSQL timezone with the user specified time zone, or the host system time zone if no time zone is configured. For example, when selecting a default timezone, Greenplum uses an algorithm to select a PostgreSQL timezone based on the host system timezone files. If the system timezone includes leap second information, Greenplum Database cannot match the system timezone with a PostgreSQL timezone. In this case, Greenplum Database calculates a "best match" with a PostgreSQL timezone based on information from the host system.

As a best practice, configure Greenplum Database and the host systems to use a known, supported timezone. This sets the timezone for the Greenplum Database master and segment instances, and prevents Greenplum Database from recalculating a "best match" timezone each time the cluster is restarted, using the current system timezone and Greenplum timezone files (which may have been updated from the IANA database since the last restart). Use the `gpconfig` utility to show and set the Greenplum Database timezone. For example, these commands show the Greenplum Database timezone and set the timezone to `US/Pacific`.

```
# gpconfig -s TimeZone
# gpconfig -c TimeZone -v 'US/Pacific'
```

You must restart Greenplum Database after changing the timezone. The command `gpstop -ra` restarts Greenplum Database. The catalog view `pg_timezone_names` provides Greenplum Database timezone information.

File System

XFS is the file system used for Greenplum Database data directories. On RHEL/CentOS systems, mount XFS volumes with the following mount options:

```
rw,nodev,noatime,nobarrier,inode64
```

The `nobarrier` option is not supported on Ubuntu systems. Use only the options:

```
rw,nodev,noatime,inode64
```

See the *recommended OS parameter settings* in the *Greenplum Database Installation Guide* for further details.

Port Configuration

Set up `ip_local_port_range` so it does not conflict with the Greenplum Database port ranges. For example, setting this range in `/etc/sysctl.conf`:

```
net.ipv4.ip_local_port_range = 10000  65535
```

you could set the Greenplum Database base port numbers to these values.

```
PORT_BASE = 6000
MIRROR_PORT_BASE = 7000
```

See the *Recommended OS Parameters Settings* in the *Greenplum Database Installation Guide* for further details.

I/O Configuration

Set the blockdev read-ahead size to 16384 on the devices that contain data directories. This command sets the read-ahead size for `/dev/sdb`.

```
# /sbin/blockdev --setra 16384 /dev/sdb
```

This command returns the read-ahead size for `/dev/sdb`.

```
# /sbin/blockdev --getra /dev/sdb
16384
```

See the *Recommended OS Parameters Settings* in the *Greenplum Database Installation Guide* for further details.

The deadline IO scheduler should be set for all data directory devices.

```
 # cat /sys/block/sdb/queue/scheduler
 noop anticipatory [deadline] cfq
```

The maximum number of OS files and processes should be increased in the `/etc/security/limits.conf` file.

```
* soft  nofile 524288
* hard  nofile 524288
* soft  nproc 131072
* hard  nproc 131072
```

Enable core files output to a known location and make sure `limits.conf` allows core files.

```
kernel.core_pattern = /var/core/core.%h.%t
# grep core /etc/security/limits.conf
* soft  core unlimited
```

OS Memory Configuration

The Linux sysctl `vm.overcommit_memory` and `vm.overcommit_ratio` variables affect how the operating system manages memory allocation. See the *`/etc/sysctl.conf`* file parameters guidelines in the *Greenplum Datatabase Installation Guide* for further details.

`vm.overcommit_memory` determines the method the OS uses for determining how much memory can be allocated to processes. This should be always set to 2, which is the only safe setting for the database.

> **Note:** For information on configuration of overcommit memory, refer to:
>
> - *https://en.wikipedia.org/wiki/Memory_overcommitment*
> - *https://www.kernel.org/doc/Documentation/vm/overcommit-accounting*

`vm.overcommit_ratio` is the percent of RAM that is used for application processes. The default is 50 on Red Hat Enterprise Linux. See *Resource Queue Segment Memory Configuration* for a formula to calculate an optimal value.

Do not enable huge pages in the operating system.

See also *Memory and Resource Management with Resource Queues*.

Shared Memory Settings

Greenplum Database uses shared memory to communicate between `postgres` processes that are part of the same `postgres` instance. The following shared memory settings should be set in `sysctl` and are rarely modified. See the *`sysctl.conf`* file parameters in the *Greenplum Database Installation Guide* for further details.

```
kernel.shmmax = 500000000
kernel.shmmni = 4096
kernel.shmall = 4000000000
```

Number of Segments per Host

Determining the number of segments to execute on each segment host has immense impact on overall system performance. The segments share the host's CPU cores, memory, and NICs with each other and with other processes running on the host. Over-estimating the number of segments a server can accommodate is a common cause of suboptimal performance.

The factors that must be considered when choosing how many segments to run per host include the following:

- Number of cores
- Amount of physical RAM installed in the server
- Number of NICs
- Amount of storage attached to server
- Mixture of primary and mirror segments
- ETL processes that will run on the hosts
- Non-Greenplum processes running on the hosts

Resource Queue Segment Memory Configuration

The `gp_vmem_protect_limit` server configuration parameter specifies the amount of memory that all active postgres processes for a single segment can consume at any given time. Queries that exceed this amount will fail. Use the following calculations to estimate a safe value for `gp_vmem_protect_limit`.

1. Calculate `gp_vmem`, the host memory available to Greenplum Database, using this formula:

   ```
   gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM)) / 1.7
   ```

   where `SWAP` is the host's swap space in GB and `RAM` is the RAM installed on the host in GB.
2. Calculate `max_acting_primary_segments`. This is the maximum number of primary segments that can be running on a host when mirror segments are activated due to a segment or host failure on another host in the cluster. With mirrors arranged in a 4-host block with 8 primary segments per host, for example, a single segment host failure would activate two or three mirror segments on each remaining host in the failed host's block. The `max_acting_primary_segments` value for this configuration is 11 (8 primary segments plus 3 mirrors activated on failure).
3. Calculate `gp_vmem_protect_limit` by dividing the total Greenplum Database memory by the maximum number of acting primaries:

   ```
   gp_vmem_protect_limit = gp_vmem / max_acting_primary_segments
   ```

   Convert to megabytes to find the value to set for the `gp_vmem_protect_limit` system configuration parameter.

For scenarios where a large number of workfiles are generated, adjust the calculation for `gp_vmem` to account for the workfiles:

```
gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM - (300KB
 * total_#_workfiles))) / 1.7
```

For information about monitoring and managing workfile usage, see the *Greenplum Database Administrator Guide*.

You can calculate the value of the `vm.overcommit_ratio` operating system parameter from the value of `gp_vmem`:

```
vm.overcommit_ratio = (RAM - 0.026 * gp_vmem) / RAM
```

See *OS Memory Configuration* for more about about `vm.overcommit_ratio`.

See also *Memory and Resource Management with Resource Queues*.

Resource Queue Statement Memory Configuration

The `statement_mem` server configuration parameter is the amount of memory to be allocated to any single query in a segment database. If a statement requires additional memory it will spill to disk. Calculate the value for `statement_mem` with the following formula:

```
(gp_vmem_protect_limit * .9) / max_expected_concurrent_queries
```

For example, for 40 concurrent queries with `gp_vmem_protect_limit` set to 8GB (8192MB), the calculation for `statement_mem` would be:

```
(8192MB * .9) / 40 = 184MB
```

Each query would be allowed 184MB of memory before it must spill to disk.

To increase `statement_mem` safely you must either increase `gp_vmem_protect_limit` or reduce the number of concurrent queries. To increase `gp_vmem_protect_limit`, you must add physical RAM and/or swap space, or reduce the number of segments per host.

Note that adding segment hosts to the cluster cannot help out-of-memory errors unless you use the additional hosts to decrease the number of segments per host.

Spill files are created when there is not enough memory to fit all the mapper output, usually when 80% of the buffer space is occupied.

Also, see *Resource Management* for best practices for managing query memory using resource queues.

Resource Queue Spill File Configuration

Greenplum Database creates *spill files* (also called *workfiles*) on disk if a query is allocated insufficient memory to execute in memory. A single query can create no more than 100,000 spill files, by default, which is sufficient for the majority of queries.

You can control the maximum number of spill files created per query and per segment with the configuration parameter `gp_workfile_limit_files_per_query`. Set the parameter to 0 to allow queries to create an unlimited number of spill files. Limiting the number of spill files permitted prevents run-away queries from disrupting the system.

A query could generate a large number of spill files if not enough memory is allocated to it or if data skew is present in the queried data. If a query creates more than the specified number of spill files, Greenplum Database returns this error:

```
ERROR: number of workfiles per query limit exceeded
```

Before raising the `gp_workfile_limit_files_per_query`, try reducing the number of spill files by changing the query, changing the data distribution, or changing the memory configuration.

The `gp_toolkit` schema includes views that allow you to see information about all the queries that are currently using spill files. This information can be used for troubleshooting and for tuning queries:

- The `gp_workfile_entries` view contains one row for each operator using disk space for workfiles on a segment at the current time. See *How to Read Explain Plans*for information about operators.
- The `gp_workfile_usage_per_query` view contains one row for each query using disk space for workfiles on a segment at the current time.
- The `gp_workfile_usage_per_segment` view contains one row for each segment. Each row displays the total amount of disk space used for workfiles on the segment at the current time.

See the *Greenplum Database Reference Guide* for descriptions of the columns in these views.

The `gp_workfile_compression` configuration parameter specifies whether the spill files are compressed. It is `off` by default. Enabling compression can improve performance when spill files are used.

Schema Design

Best practices for designing Greenplum Database schemas.

Greenplum Database is an analytical, shared-nothing database, which is much different than a highly normalized, transactional SMP database. Greenplum Database performs best with a denormalized schema design suited for MPP analytical processing, a star or snowflake schema, with large centralized fact tables connected to multiple smaller dimension tables.

Data Types

Use Types Consistently

Use the same data types for columns used in joins between tables. If the data types differ, Greenplum Database must dynamically convert the data type of one of the columns so the data values can be compared correctly. With this in mind, you may need to increase the data type size to facilitate joins to other common objects.

Choose Data Types that Use the Least Space

You can increase database capacity and improve query execution by choosing the most efficient data types to store your data.

Use `TEXT` or `VARCHAR` rather than `CHAR`. There are no performance differences among the character data types, but using `TEXT` or `VARCHAR` can decrease the storage space used.

Use the smallest numeric data type that will accommodate your data. Using `BIGINT` for data that fits in `INT` or `SMALLINT` wastes storage space.

Storage Model

Greenplum Database provides an array of storage options when creating tables. It is very important to know when to use heap storage versus append-optimized (AO) storage, and when to use row-oriented storage versus column-oriented storage. The correct selection of heap versus AO and row versus column is extremely important for large fact tables, but less important for small dimension tables.

The best practices for determining the storage model are:

1. Design and build an insert-only model, truncating a daily partition before load.
2. For large partitioned fact tables, evaluate and use optimal storage options for different partitions. One storage option is not always right for the entire partitioned table. For example, some partitions can be row-oriented while others are column-oriented.
3. When using column-oriented storage, every column is a separate file on *every* Greenplum Database segment. For tables with a large number of columns consider columnar storage for data often accessed (hot) and row-oriented storage for data not often accessed (cold).
4. Storage options should be set at the partition level.
5. Compress large tables to improve I/O performance and to make space in the cluster.

Heap Storage or Append-Optimized Storage

Heap storage is the default model, and is the model PostgreSQL uses for all database tables. Use heap storage for tables and partitions that will receive iterative `UPDATE`, `DELETE`, and singleton `INSERT` operations. Use heap storage for tables and partitions that will receive concurrent `UPDATE`, `DELETE`, and `INSERT` operations.

Use append-optimized storage for tables and partitions that are updated infrequently after the initial load and have subsequent inserts performed only in batch operations. Avoid performing singleton `INSERT`, `UPDATE`, or `DELETE` operations on append-optimized tables. Concurrent batch `INSERT` operations are acceptable, but *never* perform concurrent batch `UPDATE` or `DELETE` operations.

The append-optimized storage model is inappropriate for frequently updated tables, because space occupied by rows that are updated and deleted in append-optimized tables is not recovered and reused as efficiently as with heap tables. Append-optimized storage is intended for large tables that are loaded once, updated infrequently, and queried frequently for analytical query processing.

Row or Column Orientation

Row orientation is the traditional way to store database tuples. The columns that comprise a row are stored on disk contiguously, so that an entire row can be read from disk in a single I/O.

Column orientation stores column values together on disk. A separate file is created for each column. If the table is partitioned, a separate file is created for each column and partition. When a query accesses only a small number of columns in a column-oriented table with many columns, the cost of I/O is substantially reduced compared to a row-oriented table; any columns not referenced do not have to be retrieved from disk.

Row-oriented storage is recommended for transactional type workloads with iterative transactions where updates are required and frequent inserts are performed. Use row-oriented storage when selects against the table are wide, where many columns of a single row are needed in a query. If the majority of columns in the `SELECT` list or `WHERE` clause is selected in queries, use row-oriented storage. Use row-oriented storage for general purpose or mixed workloads, as it offers the best combination of flexibility and performance.

Column-oriented storage is optimized for read operations but it is not optimized for write operations; column values for a row must be written to different places on disk. Column-oriented tables can offer optimal query performance on large tables with many columns where only a small subset of columns are accessed by the queries.

Another benefit of column orientation is that a collection of values of the same data type can be stored together in less space than a collection of mixed type values, so column-oriented tables use less disk space (and consequently less disk I/O) than row-oriented tables. Column-oriented tables also compress better than row-oriented tables.

Use column-oriented storage for data warehouse analytic workloads where selects are narrow or aggregations of data are computed over a small number of columns. Use column-oriented storage for tables that have single columns that are regularly updated without modifying other columns in the row. Reading a complete row in a wide columnar table requires more time than reading the same row from a row-oriented table. It is important to understand that each column is a separate physical file on *every* segment in Greenplum Database.

Compression

Greenplum Database offers a variety of options to compress append-optimized tables and partitions. Use compression to improve I/O across the system by allowing more data to be read with each disk read operation. The best practice is to set the column compression settings at the partition level.

Note that new partitions added to a partitioned table do not automatically inherit compression defined at the table level; you must *specifically* define compression when you add new partitions.

Run-length encoding (RLE) compression provides the best levels of compression. Higher levels of compression usually result in more compact storage on disk, but require additional time and CPU cycles when compressing data on writes and uncompressing on reads. Sorting data, in combination with the various compression options, can achieve the highest level of compression.

Data compression should never be used for data that is stored on a compressed file system.

Test different compression types and ordering methods to determine the best compression for your specific data. For example, you might start zstd compression at level 8 or 9 and adjust for best results. RLE compression works best with files that contain repetitive data.

Distributions

An optimal distribution that results in evenly distributed data is the most important factor in Greenplum Database. In an MPP shared nothing environment overall response time for a query is measured by the completion time for all segments. The system is only as fast as the slowest segment. If the data is skewed, segments with more data will take more time to complete, so every segment must have an approximately equal number of rows and perform approximately the same amount of processing. Poor performance and out of memory conditions may result if one segment has significantly more data to process than other segments.

Consider the following best practices when deciding on a distribution strategy:

- Explicitly define a column or random distribution for all tables. Do not use the default.
- Ideally, use a single column that will distribute data across all segments evenly.
- Do not distribute on columns that will be used in the `WHERE` clause of a query.
- Do not distribute on dates or timestamps.
- The distribution key column data should contain unique values or very high cardinality.
- If a single column cannot achieve an even distribution, use a multi-column distribution key with a maximum of two columns. Additional column values do not typically yield a more even distribution and they require additional time in the hashing process.
- If a two-column distribution key cannot achieve an even distribution of data, use a random distribution. Multi-column distribution keys in most cases require motion operations to join tables, so they offer no advantages over a random distribution.

Greenplum Database random distribution is not round-robin, so there is no guarantee of an equal number of records on each segment. Random distributions typically fall within a target range of less than ten percent variation.

Optimal distributions are critical when joining large tables together. To perform a join, matching rows must be located together on the same segment. If data is not distributed on the same join column, the rows needed from one of the tables are dynamically redistributed to the other segments. In some cases a broadcast motion, in which each segment sends its individual rows to all other segments, is performed rather than a redistribution motion, where each segment rehashes the data and sends the rows to the appropriate segments according to the hash key.

Local (Co-located) Joins

Using a hash distribution that evenly distributes table rows across all segments and results in local joins can provide substantial performance gains. When joined rows are on the same segment, much of the processing can be accomplished within the segment instance. These are called *local* or *co-located* joins. Local joins minimize data movement; each segment operates independently of the other segments, without network traffic or communications between segments.

To achieve local joins for large tables commonly joined together, distribute the tables on the same column. Local joins require that both sides of a join be distributed on the same columns (and in the same order) *and* that all columns in the distribution clause are used when joining tables. The distribution columns must also be the same data type—although some values with different data types may appear to have the same representation, they are stored differently and hash to different values, so they are stored on different segments.

Data Skew

Data skew is often the root cause of poor query performance and out of memory conditions. Skewed data affects scan (read) performance, but it also affects all other query execution operations, for instance, joins and group by operations.

It is very important to *validate* distributions to *ensure* that data is evenly distributed after the initial load. It is equally important to *continue* to validate distributions after incremental loads.

The following query shows the number of rows per segment as well as the variance from the minimum and maximum numbers of rows:

```
SELECT 'Example Table' AS "Table Name",
    max(c) AS "Max Seg Rows", min(c) AS "Min Seg Rows",
    (max(c)-min(c))*100.0/max(c) AS "Percentage Difference Between Max &
 Min"
FROM (SELECT count(*) c, gp_segment_id FROM facts GROUP BY 2) AS a;
```

The `gp_toolkit` schema has two views that you can use to check for skew.

- The `gp_toolkit.gp_skew_coefficients` view shows data distribution skew by calculating the coefficient of variation (CV) for the data stored on each segment. The `skccoeff` column shows the coefficient of variation (CV), which is calculated as the standard deviation divided by the average. It takes into account both the average and variability around the average of a data series. The lower the value, the better. Higher values indicate greater data skew.
- The `gp_toolkit.gp_skew_idle_fractions` view shows data distribution skew by calculating the percentage of the system that is idle during a table scan, which is an indicator of computational skew. The `siffraction` column shows the percentage of the system that is idle during a table scan. This is an indicator of uneven data distribution or query processing skew. For example, a value of 0.1 indicates 10% skew, a value of 0.5 indicates 50% skew, and so on. Tables that have more than10% skew should have their distribution policies evaluated.

Processing Skew

Processing skew results when a disproportionate amount of data flows to, and is processed by, one or a few segments. It is often the culprit behind Greenplum Database performance and stability issues. It can happen with operations such join, sort, aggregation, and various OLAP operations. Processing skew happens in flight while a query is executing and is not as easy to detect as data skew, which is caused by uneven data distribution due to the wrong choice of distribution keys. Data skew is present at the table level, so it can be easily detected and avoided by choosing optimal distribution keys.

If single segments are failing, that is, not all segments on a host, it may be a processing skew issue. Identifying processing skew is currently a manual process. First look for spill files. If there is skew, but not enough to cause spill, it will not become a performance issue. If you determine skew exists, then find the query responsible for the skew. Following are the steps and commands to use. (Change names like the host file name passed to `gpssh` accordingly):

1. Find the OID for the database that is to be monitored for skew processing:

   ```
   SELECT oid, datname FROM pg_database;
   ```

   Example output:

   ```
     oid  |  datname
   -------+-----------
    17088 | gpadmin
    10899 | postgres
        1 | template1
    10898 | template0
    38817 | pws
    39682 | gpperfmon
   ```

```
(6 rows)
```

2. Run a `gpssh` command to check file sizes across all of the segment nodes in the system. Replace `<OID>` with the OID of the database from the prior command:

```
[gpadmin@mdw kend]$ gpssh -f ~/hosts -e \
    "du -b /data[1-2]/primary/gpseg*/base/<OID>/pgsql_tmp/*" | \
    grep -v "du -b" | sort | awk -F" " '{ arr[$1] = arr[$1] + $2 ; tot =
 tot + $2 }; END \
    { for ( i in arr ) print "Segment node" i, arr[i], "bytes (" arr[i]/
(1024**3)" GB)"; \
    print "Total", tot, "bytes (" tot/(1024**3)" GB)" }' -
```

   Example output:

```
Segment node[sdw1] 2443370457 bytes (2.27557 GB)
Segment node[sdw2] 1766575328 bytes (1.64525 GB)
Segment node[sdw3] 1761686551 bytes (1.6407 GB)
Segment node[sdw4] 1780301617 bytes (1.65804 GB)
Segment node[sdw5] 1742543599 bytes (1.62287 GB)
Segment node[sdw6] 1830073754 bytes (1.70439 GB)
Segment node[sdw7] 1767310099 bytes (1.64594 GB)
Segment node[sdw8] 1765105802 bytes (1.64388 GB)
Total 14856967207 bytes (13.8366 GB)
```

   If there is a *significant and sustained* difference in disk usage, then the queries being executed should be investigated for possible skew (the example output above does not reveal significant skew). In monitoring systems, there will always be some skew, but often it is *transient* and will be *short in duration*.

3. If significant and sustained skew appears, the next task is to identify the offending query.

   The command in the previous step sums up the entire node. This time, find the actual segment directory. You can do this from the master or by logging into the specific node identified in the previous step. Following is an example run from the master.

   This example looks specifically for sort files. Not all spill files or skew situations are caused by sort files, so you will need to customize the command:

```
$ gpssh -f ~/hosts -e
    "ls -l /data[1-2]/primary/gpseg*/base/19979/pgsql_tmp/*"
    | grep -i sort | awk '{sub(/base.*tmp\//, ".../", $10); print $1,$6,
$10}' | sort -k2 -n
```

   Here is output from this command:

```
[sdw1] 288718848
      /data1/primary/gpseg2/.../pgsql_tmp_slice0_sort_17758_0001.0[sdw1]
 291176448
      /data2/primary/gpseg5/.../pgsql_tmp_slice0_sort_17764_0001.0[sdw8]
 924581888
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0010.9[sdw4]
 980582400
      /data1/primary/gpseg18/.../pgsql_tmp_slice10_sort_29425_0001.0[sdw6]
 986447872
      /data2/primary/gpseg35/.../pgsql_tmp_slice10_sort_29602_0001.0...
[sdw5] 999620608
      /data1/primary/gpseg26/.../pgsql_tmp_slice10_sort_28637_0001.0[sdw2]
 999751680
      /data2/primary/gpseg9/.../pgsql_tmp_slice10_sort_3969_0001.0[sdw3]
 1000112128
      /data1/primary/gpseg13/.../pgsql_tmp_slice10_sort_24723_0001.0[sdw5]
 1000898560
```

```
      /data2/primary/gpseg28/.../pgsql_tmp_slice10_sort_28641_0001.0...
[sdw8] 1008009216
      /data1/primary/gpseg44/.../pgsql_tmp_slice10_sort_15671_0001.0[sdw5]
 1008566272
      /data1/primary/gpseg24/.../pgsql_tmp_slice10_sort_28633_0001.0[sdw4]
 1009451008
      /data1/primary/gpseg19/.../pgsql_tmp_slice10_sort_29427_0001.0[sdw7]
 1011187712
      /data1/primary/gpseg37/.../pgsql_tmp_slice10_sort_18526_0001.0[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0001.0[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0002.1[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0003.2[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0004.3[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0005.4[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0006.5[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0007.6[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0008.7[sdw8]
 1573741824
      /data2/primary/gpseg45/.../pgsql_tmp_slice10_sort_15673_0009.8
```

Scanning this output reveals that segment `gpseg45` on host `sdw8` is the culprit, as its sort files are larger than the others in the output.

4. Log in to the offending node with `ssh` and become root. Use the `lsof` command to find the PID for the process that owns one of the sort files:

```
[root@sdw8 ~]# lsof /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0002.1
COMMAND  PID    USER    FD   TYPE DEVICE  SIZE        NODE        NAME
postgres 15673  gpadmin 11u  REG  8,48    1073741824  64424546751 /data2/
primary/gpseg45/base/19979/pgsql_tmp/pgsql_tmp_slice10_sort_15673_0002.1
```

The PID, `15673`, is also part of the file name, but this may not always be the case.

5. Use the `ps` command with the PID to identify the database and connection information:

```
[root@sdw8 ~]# ps -eaf | grep 15673
gpadmin  15673 27471 28 12:05 ?        00:12:59 postgres: port 40003,
 sbaskin bdw
        172.28.12.250(21813) con699238 seg45 cmd32 slice10 MPPEXEC SELECT
root     29622 29566  0 12:50 pts/16   00:00:00 grep 15673
```

6. On the master, check the `pg_log` log file for the user in the previous command (`sbaskin`), connection (`con699238`, and command (`cmd32`). The line in the log file with these three values *should* be the line that contains the query, but occasionally, the command number may differ slightly. For example, the `ps` output may show `cmd32`, but in the log file it is `cmd34`. If the query is still running, the last query for the user and connection is the offending query.

The remedy for processing skew in almost all cases is to rewrite the query. Creating temporary tables can eliminate skew. Temporary tables can be randomly distributed to force a two-stage aggregation.

Partitioning

A good partitioning strategy reduces the amount of data to be scanned by reading only the partitions needed to satisfy a query.

Each partition is a separate physical file or set of tiles (in the case of column-oriented tables) on *every* segment. Just as reading a complete row in a wide columnar table requires more time than reading the same row from a heap table, *reading all partitions in a partitioned table requires more time than reading the same data from a non-partitioned table*.

Following are partitioning best practices:

- Partition large tables only, do not partition small tables.
- Use partitioning on large tables *only* when partition elimination (partition pruning) can be achieved based on query criteria and is accomplished by partitioning the table based on the query predicate. Whenever possible, use range partitioning instead of list partitioning.
- The query planner can selectively scan partitioned tables only when the query contains a direct and simple restriction of the table using immutable operators, such as =, < , <= , >, >=, and <>.
- Selective scanning recognizes `STABLE` and `IMMUTABLE` functions, but does not recognize `VOLATILE` functions within a query. For example, `WHERE` clauses such as

  ```
  date > CURRENT_DATE
  ```

  cause the query planner to selectively scan partitioned tables, but a `WHERE` clause such as

  ```
  time > TIMEOFDAY
  ```

  does not. It is important to validate that queries are selectively scanning partitioned tables (partitions are being eliminated) by examining the query `EXPLAIN` plan.
- Do not use default partitions. The default partition is always scanned but, more importantly, in many environments they tend to overfill resulting in poor performance.
- *Never* partition and distribute tables on the same column.
- Do not use multi-level partitioning. While sub-partitioning is supported, it is not recommended because typically subpartitions contain little or no data. It is a myth that performance increases as the number of partitions or subpartitions increases; the administrative overhead of maintaining many partitions and subpartitions will outweigh any performance benefits. For performance, scalability and manageability, balance partition scan performance with the number of overall partitions.
- Beware of using too many partitions with column-oriented storage.
- Consider workload concurrency and the average number of partitions opened and scanned for all concurrent queries.

Number of Partition and Columnar Storage Files

The only hard limit for the number of files Greenplum Database supports is the operating system's open file limit. It is important, however, to consider the total number of files in the cluster, the number of files on every segment, and the total number of files on a host. In an MPP shared nothing environment, every node operates independently of other nodes. Each node is constrained by its disk, CPU, and memory. CPU and I/O constraints are not common with Greenplum Database, but memory is often a limiting factor because the query execution model optimizes query performance in memory.

The optimal number of files per segment also varies based on the number of segments on the node, the size of the cluster, SQL access, concurrency, workload, and skew. There are generally six to eight segments per host, but large clusters should have fewer segments per host. When using partitioning and columnar storage it is important to balance the total number of files in the cluster, but it is *more* important to consider the number of files per segment and the total number of files on a node.

Example with 64GB Memory per Node

- Number of nodes: 16
- Number of segments per node: 8
- Average number of files per segment: 10,000

The total number of files per node is `8*10,000 = 80,000` and the total number of files for the cluster is `8*16*10,000 = 1,280,000`. The number of files increases quickly as the number of partitions and the number of columns increase.

As a general best practice, limit the total number of files per node to under 100,000. As the previous example shows, the optimal number of files per segment and total number of files per node depends on the hardware configuration for the nodes (primarily memory), size of the cluster, SQL access, concurrency, workload and skew.

Indexes

Indexes are not generally needed in Greenplum Database. Most analytical queries operate on large volumes of data, while indexes are intended for locating single rows or small numbers of rows of data. In Greenplum Database, a sequential scan is an efficient method to read data as each segment contains an equal portion of the data and all segments work in parallel to read the data.

If adding an index does not produce performance gains, drop it. Verify that every index you create is used by the optimizer.

For queries with high selectivity, indexes may improve query performance. Create an index on a single column of a columnar table for drill through purposes for high cardinality columns that are required for highly selective queries.

Do not index columns that are frequently updated. Creating an index on a column that is frequently updated increases the number of writes required on updates.

Indexes on expressions should be used only if the expression is used frequently in queries.

An index with a predicate creates a partial index that can be used to select a small number of rows from large tables.

Avoid overlapping indexes. Indexes that have the same leading column are redundant.

Indexes can improve performance on compressed append-optimized tables for queries that return a targeted set of rows. For compressed data, an index access method means only the necessary pages are uncompressed.

Create selective B-tree indexes. Index selectivity is a ratio of the number of distinct values a column has divided by the number of rows in a table. For example, if a table has 1000 rows and a column has 800 distinct values, the selectivity of the index is 0.8, which is considered good.

As a general rule, drop indexes before loading data into a table. The load will run an order of magnitude faster than loading data into a table with indexes. After the load, re-create the indexes.

Bitmap indexes are suited for querying and not updating. Bitmap indexes perform best when the column has a low cardinality—100 to 100,000 distinct values. Do not use bitmap indexes for unique columns, very high, or very low cardinality data. Do not use bitmap indexes for transactional workloads.

If indexes are needed on partitioned tables, the index columns must be different than the partition columns. A benefit of indexing partitioned tables is that because the b-tree performance degrades exponentially as the size of the b-tree grows, creating indexes on partitioned tables creates smaller b-trees that perform better than with non-partitioned tables.

Column Sequence and Byte Alignment

For optimum performance lay out the columns of a table to achieve data type byte alignment. Lay out the columns in heap tables in the following order:

1. Distribution and partition columns
2. Fixed numeric types
3. Variable data types

Lay out the data types from largest to smallest, so that `BIGINT` and `TIMESTAMP` come before `INT` and `DATE`, and all of these types come before `TEXT`, `VARCHAR`, or `NUMERIC(x,y)`. For example, 8-byte types first (`BIGINT`, `TIMESTAMP`), 4-byte types next (`INT`, `DATE`), 2-byte types next (`SMALLINT`), and variable data type last (`VARCHAR`).

Instead of defining columns in this sequence:

`Int`, `Bigint`, `Timestamp`, `Bigint`, `Timestamp`, `Int` (distribution key), `Date` (partition key), `Bigint`, `Smallint`

define the columns in this sequence:

`Int` (distribution key), `Date` (partition key), `Bigint`, `Bigint`, `Timestamp`, `Bigint`, `Timestamp`, `Int`, `Smallint`

Memory and Resource Management with Resource Groups

Managing Greenplum Database resources with resource groups.

Memory, CPU, and concurrent transaction management have a significant impact on performance in a Greenplum Database cluster. Resource groups are a newer resource management scheme that enforce memory, CPU, and concurrent transaction limits in Greenplum Database.

Configuring Memory for Greenplum Database

While it is not always possible to increase system memory, you can avoid many out-of-memory conditions by configuring resource groups to manage expected workloads.

The following operating system and Greenplum Database memory settings are significant when you manage Greenplum Database resources with resource groups:

- **vm.overcommit_memory**

  This Linux kernel parameter, set in `/etc/sysctl.conf`, identifies the method that the operating system uses to determine how much memory can be allocated to processes. `vm.overcommit_memory` must always be set to 2 for Greenplum Database systems.
- **vm.overcommit_ratio**

  This Linux kernel parameter, set in `/etc/sysctl.conf`, identifies the percentage of RAM that is used for application processes; the remainder is reserved for the operating system. Tune the setting as necessary. If your memory utilization is too low, increase the value; if your memory or swap usage is too high, decrease the setting.
- **gp_resource_group_memory_limit**

  The percentage of system memory to allocate to Greenplum Database. The default value is .7 (70%).
- **gp_workfile_limit_files_per_query**

  Set `gp_workfile_limit_files_per_query` to limit the maximum number of temporary spill files (workfiles) allowed per query. Spill files are created when a query requires more memory than it is allocated. When the limit is exceeded the query is terminated. The default is zero, which allows an unlimited number of spill files and may fill up the file system.
- **gp_workfile_compression**

  If there are numerous spill files then set `gp_workfile_compression` to compress the spill files. Compressing spill files may help to avoid overloading the disk subsystem with IO operations.
- **memory_spill_ratio**

  Set `memory_spill_ratio` to increase or decrease the amount of query operator memory Greenplum Database allots to a query. When `memory_spill_ratio` is larger than 0, it represents the percentage of resource group memory to allot to query operators. If concurrency is high, this memory amount may be small even when `memory_spill_ratio` is set to the max value of 100. When you set `memory_spill_ratio` to 0, Greenplum Database uses the `statement_mem` setting to determine the initial amount of query operator memory to allot.
- **statement_mem**

When `memory_spill_ratio` is 0, Greenplum Database uses the `statement_mem` setting to determine the amount of memory to allocate to a query.

Other considerations:

- Do not configure the operating system to use huge pages. See the *Recommended OS Parameters Settings* in the *Greenplum Installation Guide*.
- When you configure resource group memory, consider memory requirements for mirror segments that become primary segments during a failure to ensure that database operations can continue when primary segments or segment hosts fail.

Memory Considerations when using Resource Groups

Available memory for resource groups may be limited on systems that use low or no swap space, and that use the default `vm.overcommit_ratio` and `gp_resource_group_memory_limit` settings. To ensure that Greenplum Database has a reasonable per-segment-host memory limit, you may be required to increase one or more of the following configuration settings:

1. The swap size on the system.
2. The system's `vm.overcommit_ratio` setting.
3. The resource group `gp_resource_group_memory_limit` setting.

Configuring Resource Groups

Greenplum Database resource groups provide a powerful mechanism for managing the workload of the cluster. Consider these general guidelines when you configure resource groups for your system:

- A transaction submitted by any Greenplum Database role with SUPERUSER privileges runs under the default resource group named `admin_group`. Keep this in mind when scheduling and running Greenplum administration utilities.
- Ensure that you assign each non-admin role a resource group. If you do not assign a resource group to a role, queries submitted by the role are handled by the default resource group named `default_group`.
- Use the `CONCURRENCY` resource group parameter to limit the number of active queries that members of a particular resource group can run concurrently.
- Use the `MEMORY_LIMIT` and `MEMORY_SPILL_RATIO` parameters to control the maximum amount of memory that queries running in the resource group can consume.
- Greenplum Database assigns unreserved memory (100 - (sum of all resource group `MEMORY_LIMIT`s) to a global shared memory pool. This memory is available to all queries on a first-come, first-served basis.
- Alter resource groups dynamically to match the real requirements of the group for the workload and the time of day.
- Use the `gp_toolkit` views to examine resource group resource usage and to monitor how the groups are working.
- Consider using Pivotal Greenplum Command Center to create and manage resource groups, and to define the criteria under which Command Center dynamically assigns a transaction to a resource group.

Low Memory Queries

A low `statement_mem` setting (for example, in the 10MB range) has been shown to increase the performance of queries with low memory requirements. Use the `memory_spill_ratio` and `statement_mem` server configuration parameters to override the setting on a per-query basis. For example:

```
SET memory_spill_ratio=0;
SET statement_mem='10 MB';
```

Administrative Utilities and admin_group Concurrency

The default resource group for database transactions initiated by Greenplum Database SUPERUSERs is the group named `admin_group`. The default `CONCURRENCY` value for the `admin_group` resource group is 10.

Certain Greenplum Database administrative utilities may use more than one `CONCURRENCY` slot at runtime, such as `gpbackup` that you invoke with the `--jobs` option. If the utility(s) you run require more concurrent transactions than that configured for `admin_group`, consider temporarily increasing the group's `MEMORY_LIMIT` and `CONCURRENCY` values to meet the utility's requirement, making sure to return these parameters back to their original settings when the utility completes.

> **Note:** Memory allocation changes that you initiate with `ALTER RESOURCE GROUP` may not take affect immediately due to resource consumption by currently running queries. Be sure to alter resource group parameters in advance of your maintenance window.

Memory and Resource Management with Resource Queues

Avoid memory errors and manage Greenplum Database resources.

> **Note:** Resource groups are a newer resource management scheme that enforces memory, CPU, and concurrent transaction limits in Greenplum Database. The *Managing Resources* topic provides a comparison of the resource queue and the resource group management schemes. Refer to *Using Resource Groups* for configuration and usage information for this resource management scheme.

Memory management has a significant impact on performance in a Greenplum Database cluster. The default settings are suitable for most environments. Do not change the default settings until you understand the memory characteristics and usage on your system.

Resolving Out of Memory Errors

An out of memory error message identifies the Greenplum segment, host, and process that experienced the out of memory error. For example:

```
Out of memory (seg27 host.example.com pid=47093)
VM Protect failed to allocate 4096 bytes, 0 MB available
```

Some common causes of out-of-memory conditions in Greenplum Database are:

- Insufficient system memory (RAM) available on the cluster
- Improperly configured memory parameters
- Data skew at the segment level
- Operational skew at the query level

Following are possible solutions to out of memory conditions:

- Tune the query to require less memory
- Reduce query concurrency using a resource queue
- Validate the `gp_vmem_protect_limit` configuration parameter at the database level. See calculations for the maximum safe setting in *Configuring Memory for Greenplum Database*.
- Set the memory quota on a resource queue to limit the memory used by queries executed within the resource queue
- Use a session setting to reduce the `statement_mem` used by specific queries
- Decrease `statement_mem` at the database level
- Decrease the number of segments per host in the Greenplum Database cluster. This solution requires a re-initializing Greenplum Database and reloading your data.
- Increase memory on the host, if possible. (Additional hardware may be required.)

Adding segment hosts to the cluster will not in itself alleviate out of memory problems. The memory used by each query is determined by the `statement_mem` parameter and it is set when the query is invoked. However, if adding more hosts allows decreasing the number of segments per host, then the amount of memory allocated in `gp_vmem_protect_limit` can be raised.

Low Memory Queries

A low `statement_mem` setting (for example, in the 1-3MB range) has been shown to increase the performance of queries with low memory requirements. Use the `statement_mem` server configuration parameter to override the setting on a per-query basis. For example:

```
SET statement_mem='2MB';
```

Configuring Memory for Greenplum Database

Most out of memory conditions can be avoided if memory is thoughtfully managed.

It is not always possible to increase system memory, but you can prevent out-of-memory conditions by configuring memory use correctly and setting up resource queues to manage expected workloads.

It is important to include memory requirements for mirror segments that become primary segments during a failure to ensure that the cluster can continue when primary segments or segment hosts fail.

The following are recommended operating system and Greenplum Database memory settings:

- Do not configure the OS to use huge pages.
- **vm.overcommit_memory**

  This is a Linux kernel parameter, set in `/etc/sysctl.conf` and it should always be set to 2. It determines the method the OS uses for determining how much memory can be allocated to processes and 2 is the only safe setting for Greenplum Database. Please review the sysctl parameters in the *Greenplum Database Installation Guide*.
- **vm.overcommit_ratio**

  This is a Linux kernel parameter, set in *`/etc/sysctl.conf`*. It is the percentage of RAM that is used for application processes. The remainder is reserved for the operating system. The default on Red Hat is 50.

  Setting `vm.overcommit_ratio` too high may result in not enough memory being reserved for the operating system, which can result in segment host failure or database failure. Setting the value too low reduces the amount of concurrency and query complexity that can be run by reducing the amount of memory available to Greenplum Database. When increasing the setting it is important to remember to always reserve some memory for operating system activities.

  See *Greenplum Database Memory Overview* for instructions to calculate a value for `vm.overcommit_ratio`.
- **gp_vmem_protect_limit**

  Use `gp_vmem_protect_limit` to set the maximum memory that the instance can allocate for *all* work being done in each segment database. Never set this value larger than the physical RAM on the system. If `gp_vmem_protect_limit` is too high, it is possible for memory to become exhausted on the system and normal operations may fail, causing segment failures. If `gp_vmem_protect_limit` is set to a safe lower value, true memory exhaustion on the system is prevented; queries may fail for hitting the limit, but system disruption and segment failures are avoided, which is the desired behavior.

  See *Resource Queue Segment Memory Configuration* for instructions to calculate a safe value for `gp_vmem_protect_limit`.
- **runaway_detector_activation_percent**

  Runaway Query Termination, introduced in Greenplum Database 4.3.4, prevents out of memory conditions. The `runaway_detector_activation_percent` system parameter controls the percentage of `gp_vmem_protect_limit` memory utilized that triggers termination of queries. It is set on by default at 90%. If the percentage of `gp_vmem_protect_limit` memory that is utilized for a segment exceeds the specified value, Greenplum Database terminates queries based on memory usage, beginning with the query consuming the largest amount of memory. Queries are terminated until the utilized percentage of `gp_vmem_protect_limit` is below the specified percentage.

- **statement_mem**

  Use `statement_mem` to allocate memory used for a query per segment database. If additional memory is required it will spill to disk. Set the optimal value for `statement_mem` as follows:

  ```
  (vmprotect * .9) / max_expected_concurrent_queries
  ```

  The default value of `statement_mem` is 125MB. For example, on a system that is configured with 8 segments per host, a query uses 1GB of memory on each segment server (8 segments # 125MB) with the default `statement_mem` setting. Set `statement_mem` at the session level for specific queries that require additional memory to complete. This setting works well to manage query memory on clusters with low concurrency. For clusters with high concurrency also use resource queues to provide additional control on what and how much is running on the system.
- **gp_workfile_limit_files_per_query**

  Set `gp_workfile_limit_files_per_query` to limit the maximum number of temporary spill files (workfiles) allowed per query. Spill files are created when a query requires more memory than it is allocated. When the limit is exceeded the query is terminated. The default is zero, which allows an unlimited number of spill files and may fill up the file system.
- **gp_workfile_compression**

  If there are numerous spill files then set `gp_workfile_compression` to compress the spill files. Compressing spill files may help to avoid overloading the disk subsystem with IO operations.

Configuring Resource Queues

Greenplum Database resource queues provide a powerful mechanism for managing the workload of the cluster. Queues can be used to limit both the numbers of active queries and the amount of memory that can be used by queries in the queue. When a query is submitted to Greenplum Database, it is added to a resource queue, which determines if the query should be accepted and when the resources are available to execute it.

- Associate all roles with an administrator-defined resource queue.

  Each login user (role) is associated with a single resource queue; any query the user submits is handled by the associated resource queue. If a queue is not explicitly assigned the user's queries are handed by the default queue, `pg_default`.
- Do not run queries with the gpadmin role or other superuser roles.

  Superusers are exempt from resource queue limits, therefore superuser queries always run regardless of the limits set on their assigned queue.
- Use the `ACTIVE_STATEMENTS` resource queue parameter to limit the number of active queries that members of a particular queue can run concurrently.
- Use the `MEMORY_LIMIT` parameter to control the total amount of memory that queries running through the queue can utilize. By combining the `ACTIVE_STATEMENTS` and `MEMORY_LIMIT` attributes an administrator can fully control the activity emitted from a given resource queue.

  The allocation works as follows: Suppose a resource queue, `sample_queue`, has `ACTIVE_STATEMENTS` set to 10 and `MEMORY_LIMIT` set to 2000MB. This limits the queue to approximately 2 gigabytes of memory per segment. For a cluster with 8 segments per server, the total usage per server is 16 GB for `sample_queue` (2GB * 8 segments/server). If a segment server has 64GB of RAM, there could be no more than four of this type of resource queue on the system before there is a chance of running out of memory (4 queues * 16GB per queue).

  Note that by using `STATEMENT_MEM`, individual queries running in the queue can allocate more than their "share" of memory, thus reducing the memory available for other queries in the queue.
- Resource queue priorities can be used to align workloads with desired outcomes. Queues with `MAX` priority throttle activity in all other queues until the `MAX` queue completes running all queries.

- Alter resource queues dynamically to match the real requirements of the queue for the workload and time of day. You can script an operational flow that changes based on the time of day and type of usage of the system and add `crontab` entries to execute the scripts.
- Use gptoolkit to view resource queue usage and to understand how the queues are working.

System Monitoring and Maintenance

Best practices for regular maintenance that will ensure Greenplum Database high availability and optimal performance.

Monitoring

Greenplum Database includes utilities that are useful for monitoring the system.

The `gp_toolkit` schema contains several views that can be accessed using SQL commands to query system catalogs, log files, and operating environment for system status information.

The `gp_stats_missing` view shows tables that do not have statistics and require `ANALYZE` to be run.

For additional information on `gpstate` and `gpcheckperf` refer to the *Greenplum Database Utility Guide*. For information about the `gp_toolkit` schema, see the *Greenplum Database Reference Guide*.

gpstate

The `gpstate` utility program displays the status of the Greenplum system, including which segments are down, master and segment configuration information (hosts, data directories, etc.), the ports used by the system, and mapping of primary segments to their corresponding mirror segments.

Run `gpstate -Q` to get a list of segments that are marked "down" in the master system catalog.

To get detailed status information for the Greenplum system, run `gpstate -s`.

gpcheckperf

The `gpcheckperf` utility tests baseline hardware performance for a list of hosts. The results can help identify hardware issues. It performs the following checks:

- disk I/O test – measures I/O performance by writing and reading a large file using the `dd` operating system command. It reports read and write rates in megabytes per second.
- memory bandwidth test – measures sustainable memory bandwidth in megabytes per second using the STREAM benchmark.
- network performance test – runs the `gpnetbench` network benchmark program (optionally `netperf`) to test network performance. The test is run in one of three modes: parallel pair test (`-r N`), serial pair test (`-r n`), or full-matrix test (`-r M`). The minimum, maximum, average, and median transfer rates are reported in megabytes per second.

To obtain valid numbers from `gpcheckperf`, the database system must be stopped. The numbers from `gpcheckperf` can be inaccurate even if the system is up and running with no query activity.

`gpcheckperf` requires a trusted host setup between the hosts involved in the performance test. It calls `gpssh` and `gpscp`, so these utilities must also be in your `PATH`. Specify the hosts to check individually (`-h` *`host1`* `-h` *`host2`* `...`) or with `-f` *`hosts_file`*, where *`hosts_file`* is a text file containing a list of the hosts to check. If you have more than one subnet, create a separate host file for each subnet so that you can test the subnets separately.

By default, `gpcheckperf` runs the disk I/O test, the memory test, and a serial pair network performance test. With the disk I/O test, you must use the `-d` option to specify the file systems you want to test. The following command tests disk I/O and memory bandwidth on hosts listed in the `subnet_1_hosts` file:

```
$ gpcheckperf -f subnet_1_hosts -d /data1 -d /data2 -r ds
```

The `-r` option selects the tests to run: disk I/O (`d`), memory bandwidth (`s`), network parallel pair (`N`), network serial pair test (`n`), network full-matrix test (`M`). Only one network mode can be selected per execution. See the *Greenplum Database Reference Guide* for the detailed `gpcheckperf` reference.

Monitoring with Operating System Utilities

The following Linux/UNIX utilities can be used to assess host performance:

- `iostat` allows you to monitor disk activity on segment hosts.
- `top` displays a dynamic view of operating system processes.
- `vmstat` displays memory usage statistics.

You can use `gpssh` to run utilities on multiple hosts.

Best Practices

- Implement the "Recommended Monitoring and Maintenance Tasks" in the *Greenplum Database Administrator Guide*.
- Run `gpcheckperf` at install time and periodically thereafter, saving the output to compare system performance over time.
- Use all the tools at your disposal to understand how your system behaves under different loads.
- Examine any unusual event to determine the cause.
- Monitor query activity on the system by running explain plans periodically to ensure the queries are running optimally.
- Review plans to determine whether index are being used and partition elimination is occurring as expected.

Additional Information

- `gpcheckperf` reference in the *Greenplum Database Utility Guide*.
- "Recommended Monitoring and Maintenance Tasks" in the *Greenplum Database Administrator Guide*.
- *Sustainable Memory Bandwidth in Current High Performance Computers*. John D. McCalpin. Oct 12, 1995.
- *www.netperf.org* to use `netperf`, netperf must be installed on each host you test. See `gpcheckperf` reference for more information.

Updating Statistics with ANALYZE

The most important prerequisite for good query performance is to begin with accurate statistics for the tables. Updating stastistics with the `ANALYZE` statement enables the query planner to generate optimal query plans. When a table is analyzed, information about the data is stored in the system catalog tables. If the stored information is out of date, the planner can generate inefficient plans.

Generating Statistics Selectively

Running *`ANALYZE`* with no arguments updates statistics for all tables in the database. This can be a very long-running process and it is not recommended. You should `ANALYZE` tables selectively when data has changed or use the *`analyzedb`* utility.

Running `ANALYZE` on a large table can take a long time. If it is not feasible to run `ANALYZE` on all columns of a very large table, you can generate statistics for selected columns only using `ANALYZE table(column, ...)`. Be sure to include columns used in joins, `WHERE` clauses, `SORT` clauses, `GROUP BY` clauses, or `HAVING` clauses.

For a partitioned table, you can run `ANALYZE` on just partitions that have changed, for example, if you add a new partition. Note that for partitioned tables, you can run `ANALYZE` on the parent (main) table, or on the leaf nodes—the partition files where data and statistics are actually stored. The intermediate files for sub-partitioned tables store no data or statistics, so running `ANALYZE` on them does not work. You can find the names of the partition tables in the `pg_partitions` system catalog:

```
SELECT partitiontablename from pg_partitions WHERE tablename='parent_table;
```

Improving Statistics Quality

There is a trade-off between the amount of time it takes to generate statistics and the quality, or accuracy, of the statistics.

To allow large tables to be analyzed in a reasonable amount of time, ANALYZE takes a random sample of the table contents, rather than examining every row. To increase the number of sample values for all table columns adjust the `default_statistics_target` configuration parameter. The target value ranges from 1 to 1000; the default target value is 100. The `default_statistics_target` variable applies to all columns by default, and specifies the number of values that are stored in the list of common values. A larger target may improve the quality of the query planner's estimates, especially for columns with irregular data patterns. `default_statistics_target` can be set at the master/session level and requires a reload.

When to Run ANALYZE

Run ANALYZE:

- after loading data,
- after `CREATE INDEX` operations,
- and after `INSERT`, `UPDATE`, and `DELETE` operations that significantly change the underlying data.

ANALYZE requires only a read lock on the table, so it may be run in parallel with other database activity, but do not run ANALYZE while performing loads, `INSERT`, `UPDATE`, `DELETE`, and `CREATE INDEX` operations.

Configuring Automatic Statistics Collection

The `gp_autostats_mode` configuration parameter, together with the `gp_autostats_on_change_threshold` parameter, determines when an automatic analyze operation is triggered. When automatic statistics collection is triggered, the planner adds an ANALYZE step to the query.

By default, `gp_autostats_mode` is `on_no_stats`, which triggers statistics collection for `CREATE TABLE AS SELECT`, `INSERT`, or `COPY` operations on any table that has no existing statistics.

Setting `gp_autostats_mode` to `on_change` triggers statistics collection only when the number of rows affected exceeds the threshold defined by `gp_autostats_on_change_threshold`, which has a default value of 2147483647. Operations that can trigger automatic statistics collection with `on_change` are: `CREATE TABLE AS SELECT`, `UPDATE`, `DELETE`, `INSERT`, and `COPY`.

Setting `gp_autostats_mode` to `none` disables automatics statistics collection.

For partitioned tables, automatic statistics collection is not triggered if data is inserted from the top-level parent table of a partitioned table. But automatic statistics collection *is* triggered if data is inserted directly in a leaf table (where the data is stored) of the partitioned table.

Managing Bloat in a Database

Database bloat occurs in heap tables, append-optimized tables, indexes, and system catalogs and affects database performance and disk usage. You can detect database bloat and remove it from the database.

- *About Bloat*
- *Detecting Bloat*
- *Removing Bloat from Database Tables*
- *Removing Bloat from Append-Optimized Tables*
- *Removing Bloat from Indexes*
- *Removing Bloat from System Catalogs*

About Bloat

Database bloat is disk space that was used by a table or index and is available for reuse by the database but has not been reclaimed. Bloat is created when updating tables or indexes.

Because Greenplum Database heap tables use the PostgreSQL Multiversion Concurrency Control (MVCC) storage implementation, a deleted or updated row is logically deleted from the database, but a non-visible image of the row remains in the table. These deleted rows, also called expired rows, are tracked in a free space map. Running `VACUUM` marks the expired rows as free space that is available for reuse by subsequent inserts.

It is normal for tables that have frequent updates to have a small or moderate amount of expired rows and free space that will be reused as new data is added. But when the table is allowed to grow so large that active data occupies just a small fraction of the space, the table has become significantly bloated. Bloated tables require more disk storage and additional I/O that can slow down query execution.

> **Important:**
>
> It is very important to run `VACUUM` on individual tables after large `UPDATE` and `DELETE` operations to avoid the necessity of ever running `VACUUM FULL`.

Running the `VACUUM` command regularly on tables prevents them from growing too large. If the table does become significantly bloated, the `VACUUM FULL` command must be used to compact the table data.

If the free space map is not large enough to accommodate all of the expired rows, the `VACUUM` command is unable to reclaim space for expired rows that overflowed the free space map. The disk space may only be recovered by running `VACUUM FULL`, which locks the table, creates a new table, copies the table data to the new table, and then drops old table. This is an expensive operation that can take an exceptional amount of time to complete with a large table.

> **Warning:** `VACUUM FULL` acquires an `ACCESS EXCLUSIVE` lock on tables. You should not run `VACUUM FULL <database_name>`. If you run `VACUUM FULL` on tables, run it during a time when users and applications do not require access to the tables, such as during a time of low activity, or during a maintenance window.

Detecting Bloat

The statistics collected by the `ANALYZE` statement can be used to calculate the expected number of disk pages required to store a table. The difference between the expected number of pages and the actual number of pages is a measure of bloat. The `gp_toolkit` schema provides the *`gp_bloat_diag`* view that identifies table bloat by comparing the ratio of expected to actual pages. To use it, make sure statistics are up to date for all of the tables in the database, then run the following SQL:

```
gpadmin=# SELECT * FROM gp_toolkit.gp_bloat_diag;
 bdirelid | bdinspname | bdirelname | bdirelpages | bdiexppages |
     bdidiag
----------+------------+------------+-------------+-------------
+-------------------------------------
    21488 | public     | t1         |          97 |           1 |
 significant amount of bloat suspected
(1 row)
```

The results include only tables with moderate or significant bloat. Moderate bloat is reported when the ratio of actual to expected pages is greater than four and less than ten. Significant bloat is reported when the ratio is greater than ten.

The `gp_toolkit.gp_bloat_expected_pages` view lists the actual number of used pages and expected number of used pages for each database object.

```
gpadmin=# SELECT * FROM gp_toolkit.gp_bloat_expected_pages LIMIT 5;
 btdrelid | btdrelpages | btdexppages
----------+-------------+-------------
```

```
    10789 |           1 |           1
    10794 |           1 |           1
    10799 |           1 |           1
     5004 |           1 |           1
     7175 |           1 |           1
(5 rows)
```

The `btdrelid` is the object ID of the table. The `btdrelpages` column reports the number of pages the table uses; the `btdexppages` column is the number of pages expected. Again, the numbers reported are based on the table statistics, so be sure to run `ANALYZE` on tables that have changed.

Removing Bloat from Database Tables

The `VACUUM` command adds expired rows to the free space map so that the space can be reused. When `VACUUM` is run regularly on a table that is frequently updated, the space occupied by the expired rows can be promptly reused, preventing the table file from growing larger. It is also important to run `VACUUM` before the free space map is filled. For heavily updated tables, you may need to run `VACUUM` at least once a day to prevent the table from becoming bloated.

> **Warning:** When a table is significantly bloated, it is better to run `VACUUM` before running `ANALYZE`. Analyzing a severely bloated table can generate poor statistics if the sample contains empty pages, so it is good practice to vacuum a bloated table before analyzing it.

When a table accumulates significant bloat, running the `VACUUM` command is insufficient. For small tables, running `VACUUM FULL <table_name>` can reclaim space used by rows that overflowed the free space map and reduce the size of the table file. However, a `VACUUM FULL` statement is an expensive operation that requires an `ACCESS EXCLUSIVE` lock and may take an exceptionally long and unpredictable amount of time to finish for large tables. You should run `VACUUM FULL` on tables during a time when users and applications do not require access to the tables being vacuumed, such as during a time of low activity, or during a maintenance window.

Removing Bloat from Append-Optimized Tables

Append-optimized tables are handled much differently than heap tables. Although append-optimized tables allow update, insert, and delete operations, these operations are not optimized and are not recommended with append-optimized tables. If you heed this advice and use append-optimized for *load-once/read-many* workloads, `VACUUM` on an append-optimized table runs almost instantaneously.

If you do run `UPDATE` or `DELETE` commands on an append-optimized table, expired rows are tracked in an auxiliary bitmap instead of the free space map. `VACUUM` is the only way to recover the space. Running `VACUUM` on an append-optimized table with expired rows compacts a table by rewriting the entire table without the expired rows. However, no action is performed if the percentage of expired rows in the table exceeds the value of the `gp_appendonly_compaction_threshold` configuration parameter, which is 10 (10%) by default. The threshold is checked on each segment, so it is possible that a `VACUUM` statement will compact an append-only table on some segments and not others. Compacting append-only tables can be disabled by setting the `gp_appendonly_compaction` parameter to `no`.

Removing Bloat from Indexes

The `VACUUM` command only recovers space from tables. To recover the space from indexes, recreate them using the `REINDEX` command.

To rebuild all indexes on a table run `REINDEX` *`table_name`*`;`. To rebuild a particular index, run `REINDEX` *`index_name`*`;`. `REINDEX` sets the `reltuples` and `relpages` to 0 (zero) for the index, To update those statistics, run `ANALYZE` on the table after reindexing.

Removing Bloat from System Catalogs

Greenplum Database system catalog tables are heap tables and can become bloated over time. As database objects are created, altered, or dropped, expired rows are left in the system catalogs. Using

`gpload` to load data contributes to the bloat since `gpload` creates and drops external tables. (Rather than use `gpload`, it is recommended to use `gpfdist` to load data.)

Bloat in the system catalogs increases the time require to scan the tables, for example, when creating explain plans. System catalogs are scanned frequently and if they become bloated, overall system performance is degraded.

It is recommended to run `VACUUM` on system catalog tables nightly and at least weekly. At the same time, running `REINDEX SYSTEM` removes bloat from the indexes. Alternatively, you can reindex system tables using the `reindexdb` utility with the `-s` (`--system`) option. After removing catalog bloat, run `ANALYZE` to update catalog table statistics.

These are Greenplum Database system catalog maintenance steps.

1. Perform a `REINDEX` on the system catalog tables to rebuild the system catalog indexes. This removes bloat in the indexes and improves `VACUUM` performance.

   Note: When performing `REINDEX` on the system catalog tables, locking will occur on the tables and might have an impact on currently running queries. You can schedule the `REINDEX` operation during a period of low activity to avoid disrupting ongoing business operations.
2. Perform a `VACUUM` on system catalog tables.
3. Perform an `ANALYZE` on the system catalog tables to update the table statistics.

If you are performing system catalog maintenance during a maintenance period and you need to stop a process due to time constraints, run the Greenplum Database function `pg_cancel_backend(<PID>)` to safely stop a Greenplum Database process.

The following script runs `REINDEX`, `VACUUM`, and `ANALYZE` on the system catalogs.

```
#!/bin/bash
DBNAME="<database_name>"
SYSTABLES="' pg_catalog.' || relname || ';' from pg_class a, pg_namespace b \
where a.relnamespace=b.oid and b.nspname='pg_catalog' and a.relkind='r'"

reindexdb -s -d $DBNAME
psql -tc "SELECT 'VACUUM' || $SYSTABLES" $DBNAME | psql -a $DBNAME
analyzedb -s pg_catalog -d $DBNAME
```

If the system catalogs become significantly bloated, you must run `VACUUM FULL` during a scheduled downtime period. During this period, stop all catalog activity on the system; `VACUUM FULL` takes `ACCESS EXCLUSIVE` locks against the system catalog. Running `VACUUM` regularly on system catalog tables can prevent the need for this more costly procedure.

These are steps for intensive system catalog maintenance.

1. Stop all catalog activity on the Greenplum Database system.
2. Perform a `REINDEX` on the system catalog tables to rebuild the system catalog indexes. This removes bloat in the indexes and improves `VACUUM` performance.
3. Perform a `VACUUM FULL` on the system catalog tables. See the following Note.
4. Perform an `ANALYZE` on the system catalog tables to update the catalog table statistics.

   Note: The system catalog table `pg_attribute` is usually the largest catalog table. If the `pg_attribute` table is significantly bloated, a `VACUUM FULL` operation on the table might require a significant amount of time and might need to be performed separately. The presence of both of these conditions indicate a significantly bloated `pg_attribute` table that might require a long `VACUUM FULL` time:

   - The `pg_attribute` table contains a large number of records.
   - The diagnostic message for `pg_attribute` is `significant amount of bloat` in the `gp_toolkit.gp_bloat_diag` view.

Monitoring Greenplum Database Log Files

Know the location and content of system log files and monitor them on a regular basis and not just when problems arise.

The following table shows the locations of the various Greenplum Database log files. In file paths:

- `$GPADMIN_HOME` refers to the home directory of the `gpadmin` operating system user.
- `$MASTER_DATA_DIRECTORY` refers to the master data directory on the Greenplum Database master host.
- `$GPDATA_DIR` refers to a data directory on the Greenplum Database segment host.
- *`host`* identifies the Greenplum Database segment host name.
- *`segprefix`* identifies the segment prefix.
- *`N`* identifies the segment instance number.
- *`date`* is a date in the format `YYYYMMDD`.

Path	Description
`$GPADMIN_HOME/gpAdminLogs/*`	Many different types of log files, directory on each server. `$GPADMIN_HOME` is the default location for the `gpAdminLogs/` directory. You can specify a different location when you run an administrative utility command.
`$GPADMIN_HOME/gpAdminLogs/gpinitsystem_`*`date`*`.log`	system initialization log
`$GPADMIN_HOME/gpAdminLogs/gpstart_`*`date`*`.log`	start log
`$GPADMIN_HOME/gpAdminLogs/gpstop_`*`date`*`.log`	stop log
`$GPADMIN_HOME/gpAdminLogs/gpsegstart.py_`*`host`*`:gpadmin_`*`date`*`.log`	segment host start log
`$GPADMIN_HOME/gpAdminLogs/gpsegstop.py_`*`host`*`:gpadmin_`*`date`*`.log`	segment host stop log
`$MASTER_DATA_DIRECTORY/pg_log/startup.log`, `$GPDATA_DIR/`*`segprefixN`*`/pg_log/startup.log`	segment instance start log
`$MASTER_DATA_DIRECTORY/gpperfmon/logs/gpmon.*.log`	gpperfmon logs
`$MASTER_DATA_DIRECTORY/pg_log/*.csv`, `$GPDATA_DIR/`*`segprefixN`*`/pg_log/*.csv`	master and segment database logs
`$GPDATA_DIR/mirror/`*`segprefixN`*`/pg_log/*.csv`	mirror segment database logs
`$GPDATA_DIR/primary/`*`segprefixN`*`/pg_log/*.csv`	primary segment database logs
`/var/log/messages`	Global Linux system messages

Use `gplogfilter -t` (`--trouble`) first to search the master log for messages beginning with `ERROR:`, `FATAL:`, or `PANIC:`. Messages beginning with `WARNING` may also provide useful information.

To search log files on the segment hosts, use the Greenplum `gplogfilter` utility with `gpssh` to connect to segment hosts from the master host. You can identify corresponding log entries in segment logs by the `statement_id`.

Greenplum Database can be configured to rotate database logs based on the size and/or age of the current log file. The `log_rotation_size` configuration parameter sets the size of an individual log file that triggers rotation. When the current log file size is equal to or greater than this size, the file is closed and a new log file is created. The `log_rotation_age` configuration parameter specifies the age of the current log file that triggers rotation. When the specified time has elapsed since the current log file was created, a new log file is created. The default `log_rotation_age`, `1d`, creates a new log file 24 hours after the current log file was created.

Loading Data

Description of the different ways to add data to Greenplum Database.

INSERT Statement with Column Values

A singleton `INSERT` statement with values adds a single row to a table. The row flows through the master and is distributed to a segment. This is the slowest method and is not suitable for loading large amounts of data.

COPY Statement

The PostgreSQL `COPY` statement copies data from an external file into a database table. It can insert multiple rows more efficiently than an `INSERT` statement, but the rows are still passed through the master. All of the data is copied in one command; it is not a parallel process.

Data input to the `COPY` command is from a file or the standard input. For example:

```
COPY table FROM '/data/mydata.csv' WITH CSV HEADER;
```

Use `COPY` to add relatively small sets of data, for example dimension tables with up to ten thousand rows, or one-time data loads.

Use `COPY` when scripting a process that loads small amounts of data, less than 10 thousand rows.

Since COPY is a single command, there is no need to disable autocommit when you use this method to populate a table.

You can run multiple concurrent `COPY` commands to improve performance.

External Tables

External tables provide access to data in sources outside of Greenplum Database. They can be accessed with `SELECT` statements and are commonly used with the Extract, Load, Transform (ELT) pattern, a variant of the Extract, Transform, Load (ETL) pattern that takes advantage of Greenplum Database's fast parallel data loading capability.

With ETL, data is extracted from its source, transformed outside of the database using external transformation tools, such as Informatica or Datastage, and then loaded into the database.

With ELT, Greenplum external tables provide access to data in external sources, which could be read-only files (for example, text, CSV, or XML files), Web servers, Hadoop file systems, executable OS programs, or the Greenplum `gpfdist` file server, described in the next section. External tables support SQL operations such as select, sort, and join so the data can be loaded and transformed simultaneously, or loaded into a *load table* and transformed in the database into target tables.

The external table is defined with a `CREATE EXTERNAL TABLE` statement, which has a `LOCATION` clause to define the location of the data and a `FORMAT` clause to define the formatting of the source data so that the system can parse the input data. Files use the `file://` protocol, and must reside on a segment host in a location accessible by the Greenplum superuser. The data can be spread out among the segment hosts with no more than one file per primary segment on each host. The number of files listed in the `LOCATION` clause is the number of segments that will read the external table in parallel.

External Tables with Gpfdist

The fastest way to load large fact tables is to use external tables with `gpdist`. `gpfdist` is a file server program using an HTTP protocol that serves external data files to Greenplum Database segments

in parallel. A `gpfdist` instance can serve 200 MB/second and many `gpfdist` processes can run simultaneously, each serving up a portion of the data to be loaded. When you begin the load using a statement such as `INSERT INTO <table> SELECT * FROM <external_table>`, the `INSERT` statement is parsed by the master and distributed to the primary segments. The segments connect to the `gpfdist` servers and retrieve the data in parallel, parse and validate the data, calculate a hash from the distribution key data and, based on the hash key, send the row to its destination segment. By default, each `gpfdist` instance will accept up to 64 connections from segments. With many segments and `gpfdist` servers participating in the load, data can be loaded at very high rates.

Primary segments access external files in parallel when using `gpfdist` up to the value of `gp_external_max_segments`. When optimizing `gpfdist` performance, maximize the parallelism as the number of segments increase. Spread the data evenly across as many ETL nodes as possible. Split very large data files into equal parts and spread the data across as many file systems as possible.

Run two `gpfdist` instances per file system. `gpfdist` tends to be CPU bound on the segment nodes when loading. But if, for example, there are eight racks of segment nodes, there is lot of available CPU on the segments to drive more `gpfdist` processes. Run `gpfdist` on as many interfaces as possible. Be aware of bonded NICs and be sure to start enough `gpfdist` instances to work them.

It is important to keep the work even across all these resources. The load is as fast as the slowest node. Skew in the load file layout will cause the overall load to bottleneck on that resource.

The `gp_external_max_segs` configuration parameter controls the number of segments each `gpfdist` process serves. The default is 64. You can set a different value in the `postgresql.conf` configuration file on the master. Always keep `gp_external_max_segs` and the number of `gpfdist` processes an even factor; that is, the `gp_external_max_segs` value should be a multiple of the number of `gpfdist` processes. For example, if there are 12 segments and 4 `gpfdist` processes, the planner round robins the segment connections as follows:

```
Segment 1  - gpfdist 1
Segment 2  - gpfdist 2
Segment 3  - gpfdist 3
Segment 4  - gpfdist 4
Segment 5  - gpfdist 1
Segment 6  - gpfdist 2
Segment 7  - gpfdist 3
Segment 8  - gpfdist 4
Segment 9  - gpfdist 1
Segment 10 - gpfdist 2
Segment 11 - gpfdist 3
Segment 12 - gpfdist 4
```

Drop indexes before loading into existing tables and re-create the index after loading. Creating an index on pre-existing data is faster than updating it incrementally as each row is loaded.

Run `ANALYZE` on the table after loading. Disable automatic statistics collection during loading by setting `gp_autostats_mode` to `NONE`. Run `VACUUM` after load errors to recover space.

Performing small, high frequency data loads into heavily partitioned column-oriented tables can have a high impact on the system because of the number of physical files accessed per time interval.

Gpload

`gpload` is a data loading utility that acts as an interface to the Greenplum external table parallel loading feature.

Beware of using `gpload` as it can cause catalog bloat by creating and dropping external tables. Use `gpfdist` instead, since it provides the best performance.

`gpload` executes a load using a specification defined in a YAML-formatted control file. It performs the following operations:

- Invokes `gpfdist` processes
- Creates a temporary external table definition based on the source data defined
- Executes an `INSERT`, `UPDATE`, or `MERGE` operation to load the source data into the target table in the database
- Drops the temporary external table
- Cleans up `gpfdist` processes

The load is accomplished in a single transaction.

Best Practices

- Drop any indexes on an existing table before loading data and recreate the indexes after loading. Newly creating an index is faster than updating an index incrementally as each row is loaded.
- Disable automatic statistics collection during loading by setting the `gp_autostats_mode` configuration parameter to `NONE`.
- External tables are not intended for frequent or ad hoc access.
- External tables have no statistics to inform the optimizer. You can set rough estimates for the number of rows and disk pages for the external table in the `pg_class` system catalog with a statement like the following:

  ```
  UPDATE pg_class SET reltuples=400000, relpages=400
  WHERE relname='myexttable';
  ```

- When using `gpfdist`, maximize network bandwidth by running one `gpfdist` instance for each NIC on the ETL server. Divide the source data evenly between the `gpfdist` instances.
- When using `gpload`, run as many simultaneous `gpload` instances as resources allow. Take advantage of the CPU, memory, and networking resources available to increase the amount of data that can be transferred from ETL servers to the Greenplum Database.
- Use the `SEGMENT REJECT LIMIT` clause of the `COPY` statement to set a limit for the number or percentage of rows that can have errors before the `COPY FROM` command is aborted. The reject limit is per segment; when any one segment exceeds the limit, the command is aborted and no rows are added. Use the `LOG ERRORS` clause to save error rows. If a row has errors in the formatting—for example missing or extra values, or incorrect data types—Greenplum Database stores the error information and row internally. Use the `gp_read_error_log()` built-in SQL function to access this stored information.
- If the load has errors, run `VACUUM` on the table to recover space.
- After you load data into a table, run `VACUUM` on heap tables, including system catalogs, and `ANALYZE` on all tables. It is not necessary to run `VACUUM` on append-optimized tables. If the table is partitioned, you can vacuum and analyze just the partitions affected by the data load. These steps clean up any rows from aborted loads, deletes, or updates and update statistics for the table.
- Recheck for segment skew in the table after loading a large amount of data. You can use a query like the following to check for skew:

  ```
  
  ```

- By default, `gpfdist` assumes a maximum record size of 32K. To load data records larger than 32K, you must increase the maximum row size parameter by specifying the `-m <bytes>` option on the `gpfdist` command line. If you use `gpload`, set the `MAX_LINE_LENGTH` parameter in the `gpload` control file.

  > **Note:** Integrations with Informatica Power Exchange are currently limited to the default 32K record length.

Additional Information

See the *Greenplum Database Reference Guide* for detailed instructions for loading data using `gpfdist` and `gpload`.

Security

Best practices to ensure the highest level of system security.

Basic Security Best Practices

- Secure the `gpadmin` system user. Greenplum requires a UNIX user id to install and initialize the Greenplum Database system. This system user is referred to as `gpadmin` in the Greenplum documentation. The `gpadmin` user is the default database superuser in Greenplum Database, as well as the file system owner of the Greenplum installation and its underlying data files. The default administrator account is fundamental to the design of Greenplum Database. The system cannot run without it, and there is no way to limit the access of the `gpadmin` user id. This `gpadmin` user can bypass all security features of Greenplum Database. Anyone who logs on to a Greenplum host with this user id can read, alter, or delete any data, including system catalog data and database access rights. Therefore, it is very important to secure the `gpadmin` user id and only allow essential system administrators access to it. Administrators should only log in to Greenplum as `gpadmin` when performing certain system maintenance tasks (such as upgrade or expansion). Database users should never log on as `gpadmin`, and ETL or production workloads should never run as `gpadmin`.
- Assign a distinct role to each user who logs in. For logging and auditing purposes, each user who is allowed to log in to Greenplum Database should be given their own database role. For applications or web services, consider creating a distinct role for each application or service. See "Creating New Roles (Users)" in the *Greenplum Database Administrator Guide*.
- Use groups to manage access privileges. See "Creating Groups (Role Membership)" in the *Greenplum Database Administrator Guide*.
- Limit users who have the SUPERUSER role attribute. Roles that are superusers bypass all access privilege checks in Greenplum Database, as well as resource queuing. Only system administrators should be given superuser rights. See "Altering Role Attributes" in the *Greenplum Database Administrator Guide*.

Password Strength Guidelines

To protect the network from intrusion, system administrators should verify the passwords used within an organization are strong ones. The following recommendations can strengthen a password:

- Minimum password length recommendation: At least 9 characters. MD5 passwords should be 15 characters or longer.
- Mix upper and lower case letters.
- Mix letters and numbers.
- Include non-alphanumeric characters.
- Pick a password you can remember.

The following are recommendations for password cracker software that you can use to determine the strength of a password.

- John The Ripper. A fast and flexible password cracking program. It allows the use of multiple word lists and is capable of brute-force password cracking. It is available online at *http://www.openwall.com/john/*.
- Crack. Perhaps the most well-known password cracking software, Crack is also very fast, though not as easy to use as John The Ripper. It can be found online at *http://www.crypticide.com/alecm/security/crack/c50-faq.html*.

The security of the entire system depends on the strength of the root password. This password should be at least 12 characters long and include a mix of capitalized letters, lowercase letters, special characters, and numbers. It should not be based on any dictionary word.

Password expiration parameters should be configured.

Ensure the following line exists within the file `/etc/libuser.conf` under the `[import]` section.

```
login_defs = /etc/login.defs
```

Ensure no lines in the `[userdefaults]` section begin with the following text, as these words override settings from `/etc/login.defs`:

- `LU_SHADOWMAX`
- `LU_SHADOWMIN`
- `LU_SHADOWWARNING`

Ensure the following command produces no output. Any accounts listed by running this command should be locked.

```
grep "^+:" /etc/passwd /etc/shadow /etc/group
```

Note: We strongly recommend that customers change their passwords after initial setup.

```
cd /etc
chown root:root passwd shadow group gshadow
chmod 644 passwd group
chmod 400 shadow gshadow
```

Find all the files that are world-writable and that do not have their sticky bits set.

```
find / -xdev -type d \( -perm -0002 -a ! -perm -1000 \) -print
```

Set the sticky bit (`# chmod +t {dir}`) for all the directories that result from running the previous command.

Find all the files that are world-writable and fix each file listed.

```
find / -xdev -type f -perm -0002 -print
```

Set the right permissions (`# chmod o-w {file}`) for all the files generated by running the aforementioned command.

Find all the files that do not belong to a valid user or group and either assign an owner or remove the file, as appropriate.

```
find / -xdev \( -nouser -o -nogroup \) -print
```

Find all the directories that are world-writable and ensure they are owned by either root or a system account (assuming only system accounts have a User ID lower than 500). If the command generates any output, verify the assignment is correct or reassign it to root.

```
find / -xdev -type d -perm -0002 -uid +500 -print
```

Authentication settings such as password quality, password expiration policy, password reuse, password retry attempts, and more can be configured using the Pluggable Authentication Modules (PAM) framework. PAM looks in the directory `/etc/pam.d` for application-specific configuration information. Running `authconfig` or `system-config-authentication` will re-write the PAM configuration files, destroying any manually made changes and replacing them with system defaults.

The default pam_cracklib PAM module provides strength checking for passwords. To configure pam_cracklib to require at least one uppercase character, lowercase character, digit, and special character, as recommended by the U.S. Department of Defense guidelines, edit the file /etc/pam.d/system-auth to include the following parameters in the line corresponding to password requisite pam_cracklib.so try_first_pass.

```
retry=3:
dcredit=-1. Require at least one digit
ucredit=-1. Require at least one upper case character
ocredit=-1. Require at least one special character
lcredit=-1. Require at least one lower case character
minlen-14. Require a minimum password length of 14.
```

For example:

```
password required pam_cracklib.so try_first_pass retry=3\minlen=14
 dcredit=-1 ucredit=-1 ocredit=-1 lcredit=-1
```

These parameters can be set to reflect your security policy requirements. Note that the password restrictions are not applicable to the root password.

The pam_tally2 PAM module provides the capability to lock out user accounts after a specified number of failed login attempts. To enforce password lockout, edit the file /etc/pam.d/system-auth to include the following lines:

- The first of the auth lines should include:

  ```
  auth required pam_tally2.so deny=5 onerr=fail unlock_time=900
  ```

- The first of the account lines should include:

  ```
  account required pam_tally2.so
  ```

Here, the deny parameter is set to limit the number of retries to 5 and the unlock_time has been set to 900 seconds to keep the account locked for 900 seconds before it is unlocked. These parameters may be configured appropriately to reflect your security policy requirements. A locked account can be manually unlocked using the pam_tally2 utility:

```
/sbin/pam_tally2 --user {username} -reset
```

You can use PAM to limit the reuse of recent passwords. The remember option for the pam_ unix module can be set to remember the recent passwords and prevent their reuse. To accomplish this, edit the appropriate line in /etc/pam.d/system-auth to include the remember option.

For example:

```
password sufficient pam_unix.so [ … existing_options …]
remember=5
```

You can set the number of previous passwords to remember to appropriately reflect your security policy requirements.

```
cd /etc
chown root:root passwd shadow group gshadow
chmod 644 passwd group
chmod 400 shadow gshadow
```

Encrypting Data and Database Connections

Best practices for implementing encryption and managing keys.

Encryption can be used to protect data in a Greenplum Database system in the following ways:

- Connections between clients and the master database can be encrypted with SSL. This is enabled by setting the `ssl` server configuration parameter to `on` and editing the `pg_hba.conf` file. See "Encrypting Client/Server Connections" in the *Greenplum Database Administrator Guide* for information about enabling SSL in Greenplum Database.
- Greenplum Database 4.2.1 and above allow SSL encryption of data in transit between the Greenplum parallel file distribution server, `gpfdist`, and segment hosts. See *Encrypting gpfdist Connections* for more information.
- Network communications between hosts in the Greenplum Database cluster can be encrypted using IPsec. An authenticated, encrypted VPN is established between every pair of hosts in the cluster. Check your operating system documentation for IPsec support, or consider a third-party solution such as that provided by *Zettaset*.
- The `pgcrypto` module of encryption/decryption functions protects data at rest in the database. Encryption at the column level protects sensitive information, such as passwords, Social Security numbers, or credit card numbers. See *Encrypting Data in Tables using PGP* for an example.

Best Practices

- Encryption ensures that data can be seen only by users who have the key required to decrypt the data.
- Encrypting and decrypting data has a performance cost; only encrypt data that requires encryption.
- Do performance testing before implementing any encryption solution in a production system.
- Server certificates in a production Greenplum Database system should be signed by a certificate authority (CA) so that clients can authenticate the server. The CA may be local if all clients are local to the organization.
- Client connections to Greenplum Database should use SSL encryption whenever the connection goes through an insecure link.
- A symmetric encryption scheme, where the same key is used to both encrypt and decrypt, has better performance than an asymmetric scheme and should be used when the key can be shared safely.
- Use functions from the pgcrypto module to encrypt data on disk. The data is encrypted and decrypted in the database process, so it is important to secure the client connection with SSL to avoid transmitting unencrypted data.
- Use the gpfdists protocol to secure ETL data as it is loaded into or unloaded from the database. See *Encrypting gpfdist Connections*.

Key Management

Whether you are using symmetric (single private key) or asymmetric (public and private key) cryptography, it is important to store the master or private key securely. There are many options for storing encryption keys, for example, on a file system, key vault, encrypted USB, trusted platform module (TPM), or hardware security module (HSM).

Consider the following questions when planning for key management:

- Where will the keys be stored?
- When should keys expire?
- How are keys protected?
- How are keys accessed?
- How can keys be recovered and revoked?

The Open Web Application Security Project (OWASP) provides a very comprehensive *guide to securing encryption keys*.

Encrypting Data at Rest with pgcrypto

The pgcrypto module for Greenplum Database provides functions for encrypting data at rest in the database. Administrators can encrypt columns with sensitive information, such as social security numbers or credit card numbers, to provide an extra layer of protection. Database data stored in encrypted form cannot be read by users who do not have the encryption key, and the data cannot be read directly from disk.

pgcrypto is installed by default when you install Greenplum Database. You must explicitly enable pgcrypto in each database in which you want to use the module.

pgcrypto allows PGP encryption using symmetric and asymmetric encryption. Symmetric encryption encrypts and decrypts data using the same key and is faster than asymmetric encryption. It is the preferred method in an environment where exchanging secret keys is not an issue. With asymmetric encryption, a public key is used to encrypt data and a private key is used to decrypt data. This is slower then symmetric encryption and it requires a stronger key.

Using pgcrypto always comes at the cost of performance and maintainability. It is important to use encryption only with the data that requires it. Also, keep in mind that you cannot search encrypted data by indexing the data.

Before you implement in-database encryption, consider the following PGP limitations.

- No support for signing. That also means that it is not checked whether the encryption sub-key belongs to the master key.
- No support for encryption key as master key. This practice is generally discouraged, so this limitation should not be a problem.
- No support for several subkeys. This may seem like a problem, as this is common practice. On the other hand, you should not use your regular GPG/PGP keys with pgcrypto, but create new ones, as the usage scenario is rather different.

Greenplum Database is compiled with zlib by default; this allows PGP encryption functions to compress data before encrypting. When compiled with OpenSSL, more algorithms will be available.

Because pgcrypto functions run inside the database server, the data and passwords move between pgcrypto and the client application in clear-text. For optimal security, you should connect locally or use SSL connections and you should trust both the system and database administrators.

pgcrypto configures itself according to the findings of the main PostgreSQL configure script.

When compiled with `zlib`, pgcrypto encryption functions are able to compress data before encrypting.

Pgcrypto has various levels of encryption ranging from basic to advanced built-in functions. The following table shows the supported encryption algorithms.

Table 62: Pgcrypto Supported Encryption Functions

Value Functionality	Built-in	With OpenSSL
MD5	yes	yes
SHA1	yes	yes
SHA224/256/384/512	yes	yes [6].

[6] SHA2 algorithms were added to OpenSSL in version 0.9.8. For older versions, pgcrypto will use built-in code

Value Functionality	Built-in	With OpenSSL
Other digest algorithms	no	yes [7]
Blowfish	yes	yes
AES	yes	yes[8]
DES/3DES/CAST5	no	yes
Raw Encryption	yes	yes
PGP Symmetric-Key	yes	yes
PGP Public Key	yes	yes

Creating PGP Keys

To use PGP asymmetric encryption in Greenplum Database, you must first create public and private keys and install them.

This section assumes you are installing Greenplum Database on a Linux machine with the Gnu Privacy Guard (gpg) command line tool. Pivotal recommends using the latest version of GPG to create keys. Download and install Gnu Privacy Guard (GPG) for your operating system from *https://www.gnupg.org/download/*. On the GnuPG website you will find installers for popular Linux distributions and links for Windows and Mac OS X installers.

1. As root, execute the following command and choose option 1 from the menu:

```
# gpg --gen-key
gpg (GnuPG) 2.0.14; Copyright (C) 2009 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

gpg: directory `/root/.gnupg' created
gpg: new configuration file `/root/.gnupg/gpg.conf' created
gpg: WARNING: options in `/root/.gnupg/gpg.conf' are not yet active during
 this run
gpg: keyring `/root/.gnupg/secring.gpg' created
gpg: keyring `/root/.gnupg/pubring.gpg' created
Please select what kind of key you want:
 (1) RSA and RSA (default)
 (2) DSA and Elgamal
 (3) DSA (sign only)
 (4) RSA (sign only)
Your selection? 1
```

2. Respond to the prompts and follow the instructions, as shown in this example:

```
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048) Press enter to accept default key size
Requested keysize is 2048 bits
Please specify how long the key should be valid.
 0 = key does not expire
 <n> = key expires in n days
 <n>w = key expires in n weeks
 <n>m = key expires in n months
 <n>y = key expires in n years
 Key is valid for? (0) 365
```

[7] Any digest algorithm OpenSSL supports is automatically picked up. This is not possible with ciphers, which need to be supported explicitly.

[8] AES is included in OpenSSL since version 0.9.7. For older versions, pgcrypto will use built-in code.

```
Key expires at Wed 13 Jan 2016 10:35:39 AM PST
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Real name: John Doe
Email address: jdoe@email.com
Comment:
You selected this USER-ID:
 "John Doe <jdoe@email.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.
(For this demo the passphrase is blank.)
can't connect to `/root/.gnupg/S.gpg-agent': No such file or directory
You don't want a passphrase - this is probably a *bad* idea!
I will do it anyway.  You can change your passphrase at any time,
using this program with the option "--edit-key".

We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: /root/.gnupg/trustdb.gpg: trustdb created
gpg: key 2027CC30 marked as ultimately trusted
public and secret key created and signed.

gpg:  checking the trustdbgpg:
      3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg:  depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg:  next trustdb check due at 2016-01-13
pub   2048R/2027CC30 2015-01-13 [expires: 2016-01-13]
      Key fingerprint = 7EDA 6AD0 F5E0 400F 4D45   3259 077D 725E 2027
 CC30
uid                  John Doe <jdoe@email.com>
sub   2048R/4FD2EFBB 2015-01-13 [expires: 2016-01-13]
```

3. List the PGP keys by entering the following command:

```
gpg --list-secret-keys
/root/.gnupg/secring.gpg
------------------------
sec   2048R/2027CC30 2015-01-13 [expires: 2016-01-13]
uid                  John Doe <jdoe@email.com>
ssb   2048R/4FD2EFBB 2015-01-13
```

   2027CC30 is the public key and will be used to *encrypt* data in the database. 4FD2EFBB is the private (secret) key and will be used to *decrypt* data.

4. Export the keys using the following commands:

```
# gpg -a --export 4FD2EFBB > public.key
# gpg -a --export-secret-keys 2027CC30 > secret.key
```

See the *pgcrypto* documentation for more information about PGP encryption functions.

Encrypting Data in Tables using PGP

This section shows how to encrypt data inserted into a column using the PGP keys you generated.

1. Dump the contents of the `public.key` file and then copy it to the clipboard:

```
# cat public.key
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)

mQENBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
. . .
WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8Hkk8qb5x/
HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlGWE8pvgEx
/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojDyC65KJci
Pv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=XZ8J
-----END PGP PUBLIC KEY BLOCK-----
```

2. Create a table called `userssn` and insert some sensitive data, social security numbers for Bob and Alice, in this example. Paste the public.key contents after "dearmor(".

```
CREATE TABLE userssn( ssn_id SERIAL PRIMARY KEY,
    username varchar(100), ssn bytea);

INSERT INTO userssn(username, ssn)
SELECT robotccs.username, pgp_pub_encrypt(robotccs.ssn, keys.pubkey) AS
 ssn
FROM (
        VALUES ('Alice', '123-45-6788'), ('Bob', '123-45-6799'))
            AS robotccs(username, ssn)
CROSS JOIN  (SELECT  dearmor('-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)

mQENBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
. . .
WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8Hkk8qb5x/
HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlGWE8pvgEx
/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojDyC65KJci
Pv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=XZ8J
-----END PGP PUBLIC KEY BLOCK-----' AS pubkey) AS keys;
```

3. Verify that the `ssn` column is encrypted.

```
test_db=# select * from userssn;
ssn_id   | 1
username | Alice
ssn      | \301\300L\003\235M%_O
\322\357\273\001\010\000\272\227\010\341\216\360\217C\020\261)_\367
[\227\034\313:C\354d<\337\006Q\351('\2330\031lX\263Qf
\341\262\200\3015\235\036AK\242fL+\315g\322
7u\270*\304\361\355\220\021\330"\200%\264\274}R
\213\377\363\235\366\030\023)\364!\331\303\237t\277=
f \015\004\242\231\263\225%\032\271a\001\035\277\021\375X\232\304\305/
\340\334\0131\325\344[~\362\0
37-\251\336\303\340\377_\011\275\301/MY\334\343\245\244\372y\257S
\374\230\346\277\373W\346\230\276\
017fi\226Q\307\012\326\3646\000\326\005:E\364W\252=zz\010(:\343Y
\237\257iqU\0326\350=v0\362\327\350\
```

```
315G^\027:K_9\254\362\354\215<\001\304\357\331\355\323,\302\213Fe
\265\315\232\367\254\245%(\\\373
4\254\230\331\356\006B\257\333\326H\022\013\353\216F?\023\220\370\035vH5/
\227\344b\322\227\026\362=\
42\033\322<\001}\243\224;)\030zqX\214\340\221\035\275U
\345\327\214\032\351\223c\2442\345\304K\016\
011\214\307\227\237\270\026`R\205\205a~1\263\236[\037C
\260\031\205\374\245\317\033k|\366\253\037
---------
+-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-------------------------------------------------------------------------
ssn_id   | 2
username | Bob
ssn      | \301\300L\003\235M%_O\322\357\273\001\007\377t>\345\343,
\200\256\272\300\012\033M4\265\032L
L[v\262k\244\2435\264\232B\357\370d9\375\011\002\327\235<\246\210b
\030\012\337@\226Z\361\246\032\00
7`\012c\353]\355d7\360T\335\314\367\370;X\371\350*\231\212\260B
\010#RQ0\223\253c7\0132b\355\242\233\34
1\000\370\370\366\013\022\357\005i\202~\005\\z\301o\012\230Z
\014\362\244\324&\243g\351\362\325\375
\213\032\226$\2751\256XR\346k\266\030\234\267\201vUh\004\250\337A\231\223u
\247\366/i\022\275\276\350\2
20\316\306|\203+\010\261;\232\254tp\255\243\261\373Rq;\316w
\357\006\207\374U\333\365\365\245hg\031\005
\322\347ea\220\015l\212g\337\264\336b\263\004\311\210.4\340G+\221\274D
\035\375\2216\241`\346a0\273wE\2
12\342y^\202\262|A7\202t\240\333p\345G\373\253\243oCO
\011\360\247\211\014\024{\272\271\322<\001\267
\347\240\005\213\0078\036\210\307$\317\322\311\222\035\354\006<
\266\264\004\376\251q\256\220(+\030\
3270\013c\327\272\212%\363\033\252\322\337\354\276\225\232\201\212^
\304\210\2269@\3230\370{
```

4. Extract the public.key ID from the database:

```
SELECT pgp_key_id(dearmor('-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)

mQENBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
. . .
WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8Hkk8qb5x/
HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlGWE8pvgEx
/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojDyC65KJci
Pv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=XZ8J
-----END PGP PUBLIC KEY BLOCK-----'));

pgp_key_id | 9D4D255F4FD2EFBB
```

This shows that the PGP key ID used to encrypt the `ssn` column is 9D4D255F4FD2EFBB. It is recommended to perform this step whenever a new key is created and then store the ID for tracking.

You can use this key to see which key pair was used to encrypt the data:

```
SELECT username, pgp_key_id(ssn) As key_used

               FROM userssn;

                                          username | Bob
key_used | 9D4D255F4FD2EFBB
---------+-----------------
username | Alice
key_used | 9D4D255F4FD2EFBB
```

> **Note:** Different keys may have the same ID. This is rare, but is a normal event. The client application should try to decrypt with each one to see which fits — like handling ANYKEY. See *pgp_key_id()* in the pgcrypto documentation.

5. Decrypt the data using the private key.

```
SELECT username, pgp_pub_decrypt(ssn, keys.privkey)
                 AS decrypted_ssn FROM userssn
                 CROSS JOIN
                 (SELECT dearmor('-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)

lQOYBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
vG5rJAe8PuYDSJCJ74I6w7SOH3RiRIc7IfL6xYddV42l3ctd44bl8/i71hq2UyN2
/Hbsjii2ymg7ttw3jsWAx2gP9nssDgoy8QDy/o9nNqC8EGlig96ZFnFnE6Pwbhn+
ic8MD0lK5/GAlR6Hc0ZIHf8KEcavruQlikjnABEBAAEAB/wNfjjvP1brRfjjIm/j
XwUNm+sI4v2Ur7qZC94VTukPGf67lvqcYZJuqXxvZrZ8bl6mvl65xEUiZYy7BNA8
fe0PaM4Wy+Xr94Cz2bPbWgawnRNN3GAQy4rlBTrvqQWy+kmpbd87iTjwZidZNNmx
02iSzraq41Rt0Zx21Jh4rkpF67ftmzOH0vlrS0bWOvHUeMY7tCwmdPe9HbQeDlPr
n9CllUqBn4/acTtCClWAjREZn0zXAsNixtTIPC1V+9nO9YmecMkVwNfIPkIhymAM
OPFnuZ/Dz1rCRHjNHb5j6ZyUM5zDqUVnnezktxqrOENSxm0gfMGcpxHQogUMzb7c
6UyBBADSCXHPfo/VPVtMm5p1yGrNOR2jR2rUj9+poZzD2gjkt5G/xIKRlkB4uoQl
emu27wr9dVEX7ms0nvDq58iutbQ4d0JIDlcHMeSRQZluErblB75Vj3HtImblPjpn
4Jx6SWRXPUJPGXGI87u0UoBH0Lwij7M2PW7l1ao+MLEA9jAjQwQA+sr9BKPL4Ya2
r5nE72gsbCCLowkC0rdldf1RGtobwYDMpmYZhOaRKjkOTMG6rCXJxrf6LqiN8w/L
/gNziTmch35MCq/MZzA/bN4VMPyeIlwzxVZkJLsQ7yyqX/A7ac7B7DH0KfXciEXW
MSOAJhMmklW1Q1RRNw3cnYi8w3q7X40EAL/w54FVvvPqp3+sCd86SAAapM4UO2R3
tIsuNVemMWdgNXwvK8AJsz7VreVU5yZ4B8hvCuQj1C7geaN/LXhiT8foRsJC5o71
Bf+iHC/VNEv4k4uDb4lOgnHJYYyifB1wC+nn/EnXCZYQINMia1a4M6Vqc/RIfTH4
nwkZt/89LsAiR/20HHRlc3Qga2V5IDx0ZXN0a2V5QGVtYWlsLmNvbT6JAT4EEwEC
ACgFAlS1Zf0CGwMFCQHhM4AGCwkIBwMCBhUIAgkKCwQWAgMBAh4BAheAAAoJEAd9
cl4gJ8wwbfwH/3VyVsPkQl1owRJNxvXGt1bY7BfrvU52yk+PPZYoes9UpdL3CMRk
8gAM9bx5Sk08q2UXSZLC6fFOpEW4uWgmGYf8JRoC3ooezTkmCBW8I1bU0qGetzVx
opdXLuPGCE7hVWQe9HcSntiTLxGov1mJAwO7TAoccXLbyuZh9Rf5vLoQdKzcCyOH
h5IqXaQOT100TeFeEpb9TIiwcntg3WCSU5P0DGoUAOanjDZ3KE8Qp7V74fhG1EZV
zHb8FajR62CXSHFKqpBgiNxnTOk45NbXADn4eTUXPSnwPi46qoAp9UQogsfGyB1X
DOTB2UOqhutAMECaM7VtpePv79i0Z/NfnBedA5gEVLVl/QEIANabFdQ+8QMCADOi
pM1bF/JrQt3zUoc4BTqICaxdyzAfz0tUSf/7Zro2us99GlARqLWd8EqJcl/xmfcJ
iZyUam6ZAzzFXCgnH5Y1sdtMTJZdLp5WeOjwgCWG/ZLu4wzxOFFzDkiPv9RDw6e5
MNLtJrSp4hS5o2apKdbO4Ex83O4mJYnav/rEiDDCWU4T0lhv3hSKCpke6LcwsX+7
liozp+aNmP0Ypwfi4hR3UUMP70+V1beFqW2JbVLz3lLLouHRgpCzla+PzzbEKs16
jq77vG9kqZTCIzXoWaLljuitRlfJkO3vQ9hOv/8yAnkcAmowZrIBlyFg2KBzhunY
mN2YvkUAEQEAAQAH/A7r4hDrnmzX3QU6FAzePlRB7niJtE2IEN8AufF05Q2PzKU/
c1S72WjtqMAIAgYasDkOhfhcxanTneGuFVYggKT3eSDm1RFKpRjX22m0zKdwy67B
Mu95V2Oklul6OCm8dO6+2fmkGxGqc4ZsKy+jQxtxK3HG9YxMC0dvA2v2C5N4TWi3
Utc7zh//k6IbmaLd7F1d7DXt7Hn2Qsmo8I1rtgPE8grDToomTnRUodToyejEqKyI
ORwsp8n8g2CSFaXSrEyU6HbFYXSxZealhQJGYLFOZdR0MzVtZQCn/7n+IHjupndC
Nd2a8DVx3yQS3dAmvLzhFacZdjXi31wvj0moFOkEAOCz1E63SKNNksniQ11lRMJp
gaov6Ux/zGLMstwTzNouI+Kr8/db0GlSAy1Z3UoAB4tFQXEApoX9A4AJ2KqQjqOX
```

```
cZVULenfDZaxrbb9Lid7ZnTDXKVyGTWDF7ZHavHJ4981mCW17lU11zHBB9xMlx6p
dhFvb0gdy0jSLaFMFr/JBAD0fz3RrhP7e6Xll2zdBqGthjC5S/IoKwwBgw6ri2yx
LoxqBr2pl9PotJJ/JUMPhD/LxuTcOZtYjy8PKgm5jhnBDq3Ss0kNKAY1f5EkZG9a
6I4iAX/NekqSyF+OgBfC9aCgS5RG8hYoOCbp8na5R3bgiuS8IzmVmm5OhZ4MDEwg
nQP7BzmR0p5BahpZ8r3Ada7FcK+0ZLLRdLmOYF/yUrZ53SoYCZRzU/GmtQ7LkXBh
Gjqied9Bs1MHdNUolq7GaexcjZmOWHEf6w9+9M4+vxtQq1nkIWqtaphewEmd5/nf
EP3sIY0EAE3mmiLmHLqBju+UJKMNwFNeyMTqgcg50ISH8J9FRIkBJQQYAQIADwUC
VLVl/QIbDAUJAeEzgAAKCRAHfXJeICfMMOHYCACFhInZA9uAM3TC44l+MrgMUJ3r
W9izrO48WrdTsxR8WkSNbIxJoWnYxYuLyPb/shc9k65huw2SSDkj//0fRrI61FPH
QNPSvz62WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8H
kk8qb5x/HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlG
WE8pvgEx/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojD
yC65KJciPv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=fa+6
-----END PGP PRIVATE KEY BLOCK-----') AS privkey) AS keys;

username | decrypted_ssn
----------+---------------
 Alice    | 123-45-6788
 Bob      | 123-45-6799
(2 rows)
```

If you created a key with passphrase, you may have to enter it here. However for the purpose of this example, the passphrase is blank.

Encrypting gpfdist Connections

The `gpfdists` protocol is a secure version of the `gpfdist` protocol that securely identifies the file server and the Greenplum Database and encrypts the communications between them. Using `gpfdists` protects against eavesdropping and man-in-the-middle attacks.

The `gpfdists` protocol implements client/server SSL security with the following notable features:

- Client certificates are required.
- Multilingual certificates are not supported.
- A Certificate Revocation List (CRL) is not supported.
- The TLSv1 protocol is used with the `TLS_RSA_WITH_AES_128_CBC_SHA` encryption algorithm. These SSL parameters cannot be changed.
- SSL renegotiation is supported.
- The SSL ignore host mismatch parameter is set to false.
- Private keys containing a passphrase are not supported for the `gpfdist` file server (server.key) or for the Greenplum Database (client.key).
- It is the user's responsibility to issue certificates that are appropriate for the operating system in use. Generally, converting certificates to the required format is supported, for example using the SSL Converter at *https://www.sslshopper.com/ssl-converter.html*.

A `gpfdist` server started with the `--ssl` option can only communicate with the `gpfdists` protocol. A `gpfdist` server started without the `--ssl` option can only communicate with the `gpfdist` protocol. For more detail about `gpfdist` refer to the *Greenplum Database Administrator Guide*.

There are two ways to enable the `gpfdists` protocol:

- Run `gpfdist` with the `--ssl` option and then use the `gpfdists` protocol in the `LOCATION` clause of a `CREATE EXTERNAL TABLE` statement.
- Use a YAML control file with the SSL option set to true and run `gpload`. Running `gpload` starts the `gpfdist` server with the `--ssl` option and then uses the `gpfdists` protocol.

When using gpfdists, the following client certificates must be located in the `$PGDATA/gpfdists` directory on each segment:

- The client certificate file, `client.crt`
- The client private key file, `client.key`
- The trusted certificate authorities, `root.crt`

  Important: Do not protect the private key with a passphrase. The server does not prompt for a passphrase for the private key, and loading data fails with an error if one is required.

When using `gpload` with SSL you specify the location of the server certificates in the YAML control file. When using `gpfdist` with SSL, you specify the location of the server certificates with the --ssl option.

The following example shows how to securely load data into an external table. The example creates a readable external table named `ext_expenses` from all files with the `txt` extension, using the `gpfdists` protocol. The files are formatted with a pipe (`|`) as the column delimiter and an empty space as null.

1. Run `gpfdist` with the `--ssl` option on the segment hosts.
2. Log into the database and execute the following command:

```
=# CREATE EXTERNAL TABLE ext_expenses
   ( name text, date date, amount float4, category text, desc1 text )
LOCATION ('gpfdists://etlhost-1:8081/*.txt', 'gpfdists://etlhost-2:8082/
*.txt')
FORMAT 'TEXT' ( DELIMITER '|' NULL ' ') ;
```

Tuning SQL Queries

The Greenplum Database cost-based optimizer evaluates many strategies for executing a query and chooses the least costly method.

Like other RDBMS optimizers, the Greenplum optimizer takes into account factors such as the number of rows in tables to be joined, availability of indexes, and cardinality of column data when calculating the costs of alternative execution plans. The optimizer also accounts for the location of the data, preferring to perform as much of the work as possible on the segments and to minimize the amount of data that must be transmitted between segments to complete the query.

When a query runs slower than you expect, you can view the plan the optimizer selected as well as the cost it calculated for each step of the plan. This will help you determine which steps are consuming the most resources and then modify the query or the schema to provide the optimizer with more efficient alternatives. You use the SQL `EXPLAIN` statement to view the plan for a query.

The optimizer produces plans based on statistics generated for tables. It is important to have accurate statistics to produce the best plan. See *Updating Statistics with ANALYZE* in this guide for information about updating statistics.

How to Generate Explain Plans

The `EXPLAIN` and `EXPLAIN ANALYZE` statements are useful tools to identify opportunities to improve query performance. `EXPLAIN` displays the query plan and estimated costs for a query, but does not execute the query. `EXPLAIN ANALYZE` executes the query in addition to displaying the query plan. `EXPLAIN ANALYZE` discards any output from the `SELECT` statement; however, other operations in the statement are performed (for example, `INSERT`, `UPDATE`, or `DELETE`). To use `EXPLAIN ANALYZE` on a DML statement without letting the command affect the data, explicitly use `EXPLAIN ANALYZE` in a transaction (`BEGIN; EXPLAIN ANALYZE ...; ROLLBACK;`).

`EXPLAIN ANALYZE` runs the statement in addition to displaying the plan with additional information as follows:

- Total elapsed time (in milliseconds) to run the query
- Number of workers (segments) involved in a plan node operation
- Maximum number of rows returned by the segment (and its segment ID) that produced the most rows for an operation
- The memory used by the operation
- Time (in milliseconds) it took to retrieve the first row from the segment that produced the most rows, and the total time taken to retrieve all rows from that segment.

How to Read Explain Plans

An explain plan is a report detailing the steps the Greenplum Database optimizer has determined it will follow to execute a query. The plan is a tree of nodes, read from bottom to top, with each node passing its result to the node directly above. Each node represents a step in the plan, and one line for each node identifies the operation performed in that step—for example, a scan, join, aggregation, or sort operation. The node also identifies the method used to perform the operation. The method for a scan operation, for example, may be a sequential scan or an index scan. A join operation may perform a hash join or nested loop join.

Following is an explain plan for a simple query. This query finds the number of rows in the contributions table stored at each segment.

This plan has eight nodes – Seq Scan, Sort, GroupAggregate, Result, Redistribute Motion, Sort, GroupAggregate, and finally Gather Motion. Each node contains three cost estimates: cost (in sequential page reads), the number of rows, and the width of the rows.

The cost is a two-part estimate. A cost of 1.0 is equal to one sequential disk page read. The first part of the estimate is the start-up cost, which is the cost of getting the first row. The second estimate is the total cost, the cost of getting all of the rows.

The rows estimate is the number of rows output by the plan node. The number may be lower than the actual number of rows processed or scanned by the plan node, reflecting the estimated selectivity of `WHERE` clause conditions. The total cost assumes that all rows will be retrieved, which may not always be the case (for example, if you use a `LIMIT` clause).

The width estimate is the total width, in bytes, of all the columns output by the plan node.

The cost estimates in a node include the costs of all its child nodes, so the top-most node of the plan, usually a Gather Motion, has the estimated total execution costs for the plan. This is this number that the query planner seeks to minimize.

Scan operators scan through rows in a table to find a set of rows. There are different scan operators for different types of storage. They include the following:

- Seq Scan on tables — scans all rows in the table.
- Index Scan — traverses an index to fetch the rows from the table.
- Bitmap Heap Scan — gathers pointers to rows in a table from an index and sorts by location on disk. (The operator is called a Bitmap Heap Scan, even for append-only tables.)
- Dynamic Seq Scan — chooses partitions to scan using a partition selection function.

Join operators include the following:

- Hash Join – builds a hash table from the smaller table with the join column(s) as hash key. Then scans the larger table, calculating the hash key for the join column(s) and probing the hash table to find the rows with the same hash key. Hash joins are typically the fastest joins in Greenplum Database. The Hash Cond in the explain plan identifies the columns that are joined.
- Nested Loop – iterates through rows in the larger dataset, scanning the rows in the smaller dataset on each iteration. The Nested Loop join requires the broadcast of one of the tables so that all rows in one table can be compared to all rows in the other table. It performs well for small tables or tables that are limited by using an index. It is also used for Cartesian joins and range joins. There are performance implications when using a Nested Loop join with large tables. For plan nodes that contain a Nested Loop join operator, validate the SQL and ensure that the results are what is intended. Set the `enable_nestloop` server configuration parameter to OFF (default) to favor Hash Join.
- Merge Join – sorts both datasets and merges them together. A merge join is fast for pre-ordered data, but is very rare in the real world. To favor Merge Joins over Hash Joins, set the `enable_mergejoin` system configuration parameter to ON.

Some query plan nodes specify motion operations. Motion operations move rows between segments when required to process the query. The node identifies the method used to perform the motion operation. Motion operators include the following:

- Broadcast motion – each segment sends its own, individual rows to all other segments so that every segment instance has a complete local copy of the table. A Broadcast motion may not be as optimal as a Redistribute motion, so the optimizer typically only selects a Broadcast motion for small tables. A Broadcast motion is not acceptable for large tables. In the case where data was not distributed on the join key, a dynamic redistribution of the needed rows from one of the tables to another segment is performed.
- Redistribute motion – each segment rehashes the data and sends the rows to the appropriate segments according to hash key.
- Gather motion – result data from all segments is assembled into a single stream. This is the final operation for most query plans.

Other operators that occur in query plans include the following:

- Materialize – the planner materializes a subselect once so it does not have to repeat the work for each top-level row.
- InitPlan – a pre-query, used in dynamic partition elimination, performed when the values the planner needs to identify partitions to scan are unknown until execution time.
- Sort – sort rows in preparation for another operation requiring ordered rows, such as an Aggregation or Merge Join.
- Group By – groups rows by one or more columns.
- Group/Hash Aggregate – aggregates rows using a hash.
- Append – concatenates data sets, for example when combining rows scanned from partitions in a partitioned table.
- Filter – selects rows using criteria from a `WHERE` clause.
- Limit – limits the number of rows returned.

Optimizing Greenplum Queries

This topic describes Greenplum Database features and programming practices that can be used to enhance system performance in some situations.

To analyze query plans, first identify the plan nodes where the estimated cost to perform the operation is very high. Determine if the estimated number of rows and cost seems reasonable relative to the number of rows for the operation performed.

If using partitioning, validate that partition elimination is achieved. To achieve partition elimination the query predicate (`WHERE` clause) must be the same as the partitioning criteria. Also, the `WHERE` clause must not contain an explicit value and cannot contain a subquery.

Review the execution order of the query plan tree. Review the estimated number of rows. You want the execution order to build on the smaller tables or hash join result and probe with larger tables. Optimally, the largest table is used for the final join or probe to reduce the number of rows being passed up the tree to the topmost plan nodes. If the analysis reveals that the order of execution builds and/or probes is not optimal ensure that database statistics are up to date. Running `ANALYZE` will likely address this and produce an optimal query plan.

Look for evidence of computational skew. Computational skew occurs during query execution when execution of operators such as Hash Aggregate and Hash Join cause uneven execution on the segments. More CPU and memory are used on some segments than others, resulting in less than optimal execution. The cause could be joins, sorts, or aggregations on columns that have low cardinality or non-uniform distributions. You can detect computational skew in the output of the `EXPLAIN ANALYZE` statement for a query. Each node includes a count of the maximum rows processed by any one segment and the average rows processed by all segments. If the maximum row count is much higher than the average, at least one segment has performed much more work than the others and computational skew should be suspected for that operator.

Identify plan nodes where a Sort or Aggregate operation is performed. Hidden inside an Aggregate operation is a Sort. If the Sort or Aggregate operation involves a large number of rows, there is an opportunity to improve query performance. A HashAggregate operation is preferred over Sort and Aggregate operations when a large number of rows are required to be sorted. Usually a Sort operation is chosen by the optimizer due to the SQL construct; that is, due to the way the SQL is written. Most Sort operations can be replaced with a HashAggregate if the query is rewritten. To favor a HashAggregate operation over a Sort and Aggregate operation ensure that the `enable_groupagg` server configuration parameter is set to `ON`.

When an explain plan shows a broadcast motion with a large number of rows, you should attempt to eliminate the broadcast motion. One way to do this is to use the `gp_segments_for_planner` server configuration parameter to increase the cost estimate of the motion so that alternatives are favored. The `gp_segments_for_planner` variable tells the query planner how many primary segments to use in its calculations. The default value is zero, which tells the planner to use the actual number of primary segments in estimates. Increasing the number of primary segments increases the cost

of the motion, thereby favoring a redistribute motion over a broadcast motion. For example, setting `gp_segments_for_planner = 100000` tells the planner that there are 100,000 segments. Conversely, to influence the optimizer to broadcast a table and not redistribute it, set `gp_segments_for_planner` to a low number, for example 2.

Greenplum Grouping Extensions

Greenplum Database aggregation extensions to the `GROUP BY` clause can perform some common calculations in the database more efficiently than in application or procedure code:

- `GROUP BY ROLLUP(col1, col2, col3)`
- `GROUP BY CUBE(col1, col2, col3)`
- `GROUP BY GROUPING SETS((col1, col2), (col1, col3))`

A `ROLLUP` grouping creates aggregate subtotals that roll up from the most detailed level to a grand total, following a list of grouping columns (or expressions). `ROLLUP` takes an ordered list of grouping columns, calculates the standard aggregate values specified in the `GROUP BY` clause, then creates progressively higher-level subtotals, moving from right to left through the list. Finally, it creates a grand total.

A `CUBE` grouping creates subtotals for all of the possible combinations of the given list of grouping columns (or expressions). In multidimensional analysis terms, `CUBE` generates all the subtotals that could be calculated for a data cube with the specified dimensions.

You can selectively specify the set of groups that you want to create using a `GROUPING SETS` expression. This allows precise specification across multiple dimensions without computing a whole `ROLLUP` or `CUBE`.

Refer to the *Greenplum Database Reference Guide* for details of these clauses.

Window Functions

Window functions apply an aggregation or ranking function over partitions of the result set—for example, `sum(population) over (partition by city)`. Window functions are powerful and, because they do all of the work in the database, they have performance advantages over front-end tools that produce similar results by retrieving detail rows from the database and reprocessing them.

- The `row_number()` window function produces row numbers for the rows in a partition, for example, `row_number() over (order by id)`.
- When a query plan indicates that a table is scanned in more than one operation, you may be able to use window functions to reduce the number of scans.
- It is often possible to eliminate self joins by using window functions.

High Availability

Greenplum Database supports highly available, fault-tolerant database services when you enable and properly configure Greenplum high availability features. To guarantee a required level of service, each component must have a standby ready to take its place if it should fail.

Disk Storage

With the Greenplum Database "shared-nothing" MPP architecture, the master host and segment hosts each have their own dedicated memory and disk storage, and each master or segment instance has its own independent data directory. For both reliability and high performance, Pivotal recommends a hardware RAID storage solution with from 8 to 24 disks. A larger number of disks improves I/O throughput when using RAID 5 (or 6) because striping increases parallel disk I/O. The RAID controller can continue to function with a failed disk because it saves parity data on each disk in a way that it can reconstruct the data on any failed member of the array. If a hot spare is configured (or an operator replaces the failed disk with a new one) the controller rebuilds the failed disk automatically.

RAID 1 exactly mirrors disks, so if a disk fails, a replacement is immediately available with performance equivalent to that before the failure. With RAID 5 each I/O for data on the failed array member must be reconstructed from data on the remaining active drives until the replacement disk is rebuilt, so there is a temporary performance degradation. If the Greenplum master and segments are mirrored, you can switch any affected Greenplum instances to their mirrors during the rebuild to maintain acceptable performance.

A RAID disk array can still be a single point of failure, for example, if the entire RAID volume fails. At the hardware level, you can protect against a disk array failure by mirroring the array, using either host operating system mirroring or RAID controller mirroring, if supported.

It is important to regularly monitor available disk space on each segment host. Query the `gp_disk_free` external table in the `gptoolkit` schema to view disk space available on the segments. This view runs the Linux `df` command. Be sure to check that there is sufficient disk space before performing operations that consume large amounts of disk, such as copying a large table.

See `gp_toolkit.gp_disk_free` in the *Greenplum Database Reference Guide*.

Best Practices

- Use a hardware RAID storage solution with 8 to 24 disks.
- Use RAID 1, 5, or 6 so that the disk array can tolerate a failed disk.
- Configure a hot spare in the disk array to allow rebuild to begin automatically when disk failure is detected.
- Protect against failure of the entire disk array and degradation during rebuilds by mirroring the RAID volume.
- Monitor disk utilization regularly and add additional space when needed.
- Monitor segment skew to ensure that data is distributed evenly and storage is consumed evenly at all segments.

Master Mirroring

The Greenplum Database master instance is clients' single point of access to the system. The master instance stores the global system catalog, the set of system tables that store metadata about the database instance, but no user data. If an unmirrored master instance fails or becomes inaccessible, the Greenplum instance is effectively off-line, since the entry point to the system has been lost. For this reason, a standby master must be ready to take over if the primary master fails.

Master mirroring uses two processes, a sender on the active master host and a receiver on the mirror host, to synchronize the mirror with the master. As changes are applied to the master system catalogs,

the active master streams its write-ahead log (WAL) to the mirror so that each transaction applied on the master is applied on the mirror.

The mirror is a *warm standby*. If the primary master fails, switching to the standby requires an administrative user to run the `gpactivatestandby` utility on the standby host so that it begins to accept client connections. Clients must reconnect to the new master and will lose any work that was not committed when the primary failed.

See "Enabling High Availability Features" in the *Greenplum Database Administrator Guide* for more information.

Best Practices

- Set up a standby master instance—a *mirror*—to take over if the primary master fails.
- The standby can be on the same host or on a different host, but it is best practice to place it on a different host from the primary master to protect against host failure.
- Plan how to switch clients to the new master instance when a failure occurs, for example, by updating the master address in DNS.
- Set up monitoring to send notifications in a system monitoring application or by email when the primary fails.

Segment Mirroring

Greenplum Database segment instances each store and manage a portion of the database data, with coordination from the master instance. If any unmirrored segment fails, the database may have to be shutdown and recovered, and transactions occurring after the most recent backup could be lost. Mirroring segments is, therefore, an essential element of a high availability solution.

A segment mirror is a hot standby for a primary segment. Greenplum Database detects when a segment is unavailable and automatically activates the mirror. During normal operation, when the primary segment instance is active, data is replicated from the primary to the mirror in two ways:

- The transaction commit log is replicated from the primary to the mirror before the transaction is committed. This ensures that if the mirror is activated, the changes made by the last successful transaction at the primary are present at the mirror. When the mirror is activated, transactions in the log are applied to tables in the mirror.
- Second, segment mirroring uses physical file replication to update heap tables. Greenplum Server stores table data on disk as fixed-size blocks packed with tuples. To optimize disk I/O, blocks are cached in memory until the cache fills and some blocks must be evicted to make room for newly updated blocks. When a block is evicted from the cache it is written to disk and replicated over the network to the mirror. Because of the caching mechanism, table updates at the mirror can lag behind the primary. However, because the transaction log is also replicated, the mirror remains consistent with the primary. If the mirror is activated, the activation process updates the tables with any unapplied changes in the transaction commit log.

When the acting primary is unable to access its mirror, replication stops and state of the primary changes to "Change Tracking." The primary saves changes that have not been replicated to the mirror in a system table to be replicated to the mirror when it is back on-line.

The master automatically detects segment failures and activates the mirror. Transactions in progress at the time of failure are restarted using the new primary. Depending on how mirrors are deployed on the hosts, the database system may be unbalanced until the original primary segment is recovered. For example, if each segment host has four primary segments and four mirror segments, and a mirror is activated on one host, that host will have five active primary segments. Queries are not complete until the last segment has finished its work, so performance can be degraded until the balance is restored by recovering the original primary.

Administrators perform the recovery while Greenplum Database is up and running by running the `gprecoverseg` utility. This utility locates the failed segments, verifies they are valid, and compares the

transactional state with the currently active segment to determine changes made while the segment was offline. `gprecoverseg` synchronizes the changed database files with the active segment and brings the segment back online.

It is important to reserve enough memory and CPU resources on segment hosts to allow for increased activity from mirrors that assume the primary role during a failure. The formulas provided in *Configuring Memory for Greenplum Database* for configuring segment host memory include a factor for the maximum number of primary hosts on any one segment during a failure. The arrangement of mirrors on the segment hosts affects this factor and how the system will respond during a failure. See *Segment Mirroring Configurations* for a discussion of segment mirroring options.

Best Practices

- Set up mirrors for all segments.
- Locate primary segments and their mirrors on different hosts to protect against host failure.
- Mirrors can be on a separate set of hosts or co-located on hosts with primary segments.
- Set up monitoring to send notifications in a system monitoring application or by email when a primary segment fails.
- Recover failed segments promptly, using the `gprecoverseg` utility, to restore redundancy and return the system to optimal balance.

Dual Clusters

For some use cases, an additional level of redundancy can be provided by maintaining two Greenplum Database clusters that store the same data. The decision to implement dual clusters should be made with business requirements in mind.

There are two recommended methods for keeping the data synchronized in a dual cluster configuration. The first method is called Dual ETL. ETL (extract, transform, and load) is the common data warehousing process of cleansing, transforming, validating, and loading data into a data warehouse. With Dual ETL, the ETL processes are performed twice, in parallel on each cluster, and validated each time. Dual ETL provides for a complete standby cluster with the same data. It also provides the capability to query the data on both clusters, doubling the processing throughput. The application can take advantage of both clusters as needed and also ensure that the ETL is successful and validated on both sides.

The second mechanism for maintaining dual clusters is backup and restore. The data is backedup on the primary cluster, then the backup is replicated to and restored on the second cluster. The backup and restore mechanism has higher latency than Dual ETL, but requires less application logic to be developed. Backup and restore is ideal for use cases where data modifications and ETL are done daily or less frequently.

Best Practices

- Consider a Dual Cluster configuration to provide an additional level of redundancy and additional query processing throughput.

Backup and Restore

Backups are recommended for Greenplum Database databases unless the data in the database can be easily and cleanly regenerated from source data. Backups protect from operational, software, or hardware errors.

The `gpbackup` utility makes backups in parallel across the segments, so that backups scale as the cluster grows in hardware size.

A backup strategy must consider where the backups will be written and where they will be stored. Backups can be taken to the local cluster disks, but they should not be stored there permanently. If the database and its backup are on the same storage, they can be lost simultaneously. The backup also occupies space

that could be used for database storage or operations. After performing a local backup, the files should be copied to a safe, off-cluster location.

An alternative is to back up directly to an NFS mount. If each host in the cluster has an NFS mount, the backups can be written directly to NFS storage. A scale-out NFS solution is recommended to ensure that backups do not bottleneck on the IO throughput of the NFS device. Dell EMC Isilon is an example of this type of solution and can scale alongside the Greenplum cluster.

Finally, through native API integration, Greenplum Database can stream backups directly to the Dell EMC Data Domain enterprise backup platform.

Best Practices

- Back up Greenplum databases regularly unless the data is easily restored from sources.
- Use the `gpbackup` command to specify only the schema and tables that you want backed up. See the *`gpbackup`* reference for more information.
- `gpbackup` places `SHARED ACCESS` locks on the set of tables to back up. Backups with fewer tables are more efficient for selectively restoring schemas and tables, since `gprestore` does not have to search through the entire database.
- If backups are saved to local cluster storage, move the files to a safe, off-cluster location when the backup is complete. Backup files and database files that reside on the same storage can be lost simultaneously.
- If backups are saved to NFS mounts, use a scale-out NFS solution such as Dell EMC Isilon to prevent IO bottlenecks.
- Consider using Pivotal Greenplum Database integration to stream backups to the Dell EMC Data Domain enterprise backup platform.

Detecting Failed Master and Segment Instances

Recovering from system failures requires intervention from a system administrator, even when the system detects a failure and activates a standby for the failed component. In each case, the failed component must be replaced or recovered to restore full redundancy. Until the failed component is recovered, the active component lacks a standby, and the system may not be executing optimally. For these reasons, it is important to perform recovery operations promptly. Constant system monitoring ensures that administrators are aware of failures that demand their attention.

The Greenplum Database server `ftsprobe` subprocess handles fault detection. `ftsprobe` connects to and scans all segments and database processes at intervals that you can configure with the `gp_fts_probe_interval` configuration parameter. If `ftsprobe` cannot connect to a segment, it marks the segment “down” in the Greenplum Database system catalog. The segment remains down until an administrator runs the `gprecoverseg` recovery utility.

Best Practices

- Run the `gpstate` utility to see the overall state of the Greenplum system.

Additional Information

Greenplum Database Administrator Guide:

- *Monitoring a Greenplum System*
- *Recovering a Failed Segment*

Greenplum Database Utility Guide:

- *`gpstate`* - view state of the Greenplum system
- *`gprecoverseg`* - recover a failed segment
- *`gpactivatestandby`* - make the standby master the active master

RDBMS MIB Specification

Segment Mirroring Configurations

Segment mirroring allows database queries to fail over to a backup segment if the primary segment fails or becomes unavailable. Pivotal requires mirroring for supported production Greenplum Database systems.

A primary segment and its mirror must be on different hosts to ensure high availability. Each host in a Greenplum Database system has the same number of primary segments and mirror segments. Multi-homed hosts should have the same numbers of primary and mirror segments on each interface. This ensures that segment hosts and network resources are equally loaded when all primary segments are operational and brings the most resources to bear on query processing.

When a segment becomes unavailable, its mirror segment on another host becomes the active primary and processing continues. The additional load on the host creates skew and degrades performance, but should allow the system to continue. A database query is not complete until all segments return results, so a single host with an additional active primary segment has the same effect as adding an additional primary segment to every host in the cluster.

The least amount of performance degradation in a failover scenario occurs when no host has more than one mirror assuming the primary role. If multiple segments or hosts fail, the amount of degradation is determined by the host or hosts with the largest number of mirrors assuming the primary role. Spreading a host's mirrors across the remaining hosts minimizes degradation when any single host fails.

It is important, too, to consider the cluster's tolerance for multiple host failures and how to maintain a mirror configuration when expanding the cluster by adding hosts. There is no mirror configuration that is ideal for every situation.

You can allow Greenplum Database to arrange mirrors on the hosts in the cluster using one of two standard configurations, or you can design your own mirroring configuration.

The two standard mirroring arrangements are *group mirroring* and *spread mirroring*:

- **Group mirroring** — Each host mirrors another host's primary segments. This is the default for `gpinitsystem` and `gpaddmirrors`.
- **Spread mirroring** — Mirrors are spread across the available hosts. This requires that the number of hosts in the cluster is greater than the number of segments per host.

You can design a custom mirroring configuration and use the Greenplum `gpaddmirrors` or `gpmovemirrors` utilities to set up the configuration.

Block mirroring is a custom mirror configuration that divides hosts in the cluster into equally sized blocks and distributes mirrors evenly to hosts within the block. If a primary segment fails, its mirror on another host within the same block becomes the active primary. If a segment host fails, mirror segments on each of the other hosts in the block become active.

The following sections compare the group, spread, and block mirroring configurations.

Group Mirroring

Group mirroring is easiest to set up and is the default Greenplum mirroring configuration. It is least expensive to expand, since it can be done by adding as few as two hosts. There is no need to move mirrors after expansion to maintain a consistent mirror configuration.

The following diagram shows a group mirroring configuration with eight primary segments on four hosts.

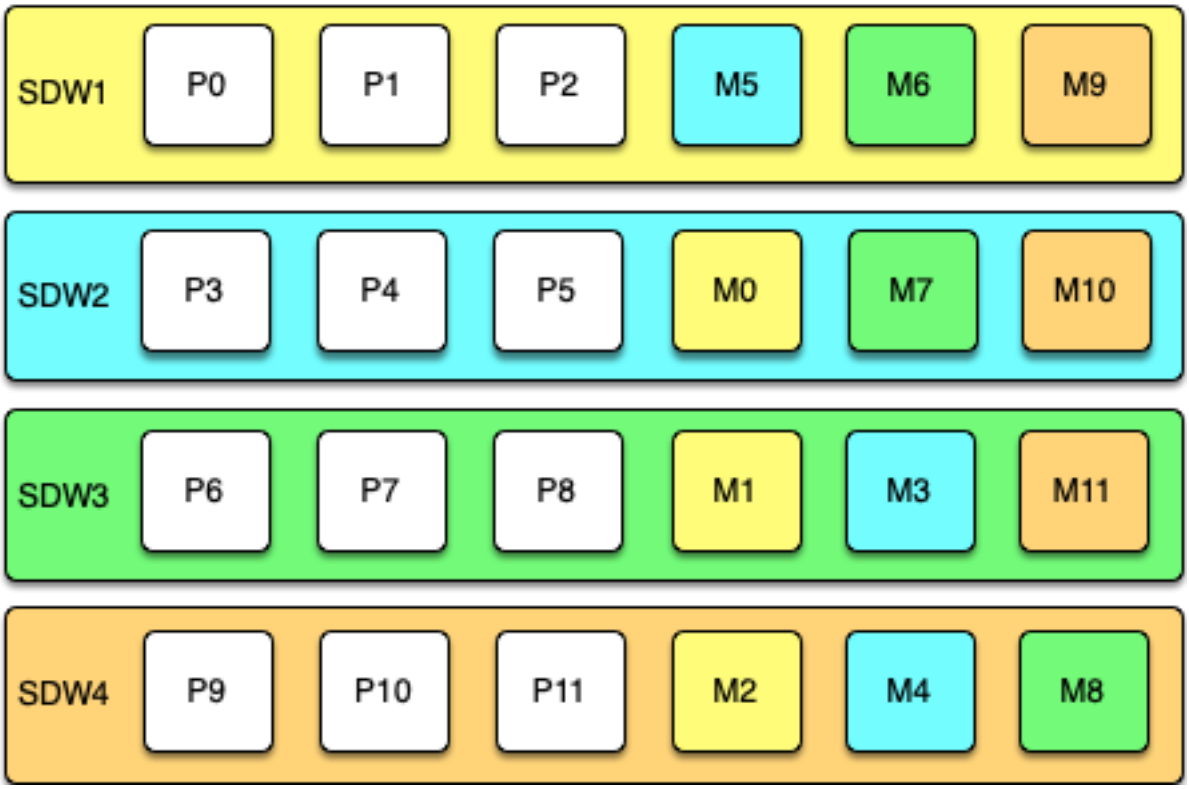
Unless both the primary and mirror of the same segment instance fail, up to half of your hosts can fail and the cluster will continue to run as long as resources (CPU, memory, and IO) are sufficient to meet the needs.

Any host failure will degrade performance by half or more because the host with the mirrors will have twice the number of active primaries. If your resource utilization is normally greater than 50%, you will have to adjust your workload until the failed host is recovered or replaced. If you normally run at less than 50% resource utilization the cluster can continue to operate at a degraded level of performance until the failure is corrected.

Spread Mirroring

With spread mirroring, mirrors for each host's primary segments are spread across as many hosts as there are segments per host. Spread mirroring is easy to set up when the cluster is initialized, but requires that the cluster have at least one more host than there are segments per host.

The following diagram shows the spread mirroring configuration for a cluster with three primaries on four hosts.



Expanding a cluster with spread mirroring requires more planning and may take more time. You must either add a set of hosts equal to the number of primaries per host plus one, or you can add two nodes in a group mirroring configuration and, when the expansion is complete, move mirrors to recreate the spread mirror configuration.

Spread mirroring has the least performance impact for a single failed host because each host's mirrors are spread across the maximum number of hosts. Load is increased by 1/*Nth*, where *N* is the number of primaries per host. Spread mirroring is, however, the most likely configuration to suffer catastrophic failure if two or more hosts fail simultaneously.

Block Mirroring

With block mirroring, nodes are divided into blocks, for example a block of four or eight hosts, and the mirrors for segments on each host are placed on other hosts within the block. Depending on the number of hosts in the block and the number of primary segments per host, each host maintains more than one mirror for each other host's segments.

The following diagram shows a single block mirroring configuration for a block of four hosts, each with eight primary segments:

If there are eight hosts, an additional four-host block is added with the mirrors for primary segments 32 through 63 set up in the same pattern.

A cluster with block mirroring is easy to expand because each block is a self-contained primary mirror group. The cluster is expanded by adding one or more blocks. There is no need to move mirrors after expansion to maintain a consistent mirror setup. This configuration is able to survive multiple host failures as long as the failed hosts are in different blocks.

Because each host in a block has multiple mirror instances for each other host in the block, block mirroring has a higher performance impact for host failures than spread mirroring, but a lower impact than group mirroring. The expected performance impact varies by block size and primary segments per node. As with group mirroring, if the resources are available, performance will be negatively impacted but the cluster will remain available. If resources are insufficient to accommodate the added load you must reduce the workload until the failed node is replaced.

Implementing Block Mirroring

Block mirroring is not one of the automatic options Greenplum Database offers when you set up or expand a cluster. To use it, you must create your own configuration.

For a new Greenplum system, you can initialize the cluster without mirrors, and then run `gpaddmirrors -i mirror_config_file` with a custom mirror configuration file to create the mirrors for each block. You must create the file system locations for the mirror segments before you run `gpaddmirrors`. See the `gpaddmirrors` reference page in the *Greenplum Database Management Utility Guide* for details.

If you expand a system that has block mirroring or you want to implement block mirroring at the same time you expand a cluster, it is recommended that you complete the expansion first, using the default grouping mirror configuration, and then use the `gpmovemirrors` utility to move mirrors into the block configuration.

To implement block mirroring with an existing system that has a different mirroring scheme, you must first determine the desired location for each mirror according to your block configuration, and then determine which of the existing mirrors must be relocated. Follow these steps:

1. Run the following query to find the current locations of the primary and mirror segments:

   ```
   SELECT dbid, content, role, port, hostname, datadir FROM
    gp_segment_configuration WHERE content > -1 ;
   ```

   The `gp_segment_configuration` system catalog table contains the current segment configuration.
2. Create a list with the current mirror location and the desired block mirroring location, then remove any mirrors from the list that are already on the correct host.
3. Create an input file for the `gpmovemirrors` utility with an entry for each mirror that must be moved.

   The `gpmovemirrors` input file has the following format:

   ```
   contentID|address|port|data_dir new_address|port|data_dir
   ```

   Where *contentID* is the segment instance content ID, *address* is the host name or IP address of the segment host, *port* is the communication port, and `data_dir` is the segment instance data directory.

   The following example `gpmovemirrors` input file specifies three mirror segments to move.

   ```
   1|sdw2|50001|/data2/mirror/gpseg1 sdw3|50001|/data/mirror/gpseg1
   2|sdw2|50001|/data2/mirror/gpseg2 sdw4|50001|/data/mirror/gpseg2
   3|sdw3|50001|/data2/mirror/gpseg3 sdw1|50001|/data/mirror/gpseg3
   ```
4. Run `gpmovemirrors` with a command like the following:

   ```
   gpmovemirrors -i mirror_config_file
   ```

The `gpmovemirrors` utility validates the input file, calls `gprecoverseg` to relocate each specified mirror, and removes the original mirror. It creates a backout configuration file which can be used as input to `gpmovemirrors` to undo the changes that were made. The backout file has the same name as the input file, with the suffix `_backout_timestamp` added.

See the *Greenplum Database Management Utility Reference* for complete information about the `gpmovemirrors` utility.

Chapter 6

Greenplum Database Utility Guide

Reference information for Greenplum Database utility programs.

About the Greenplum Database Utilities

General information about using the Greenplum Database utility programs.

Referencing IP Addresses

When you reference IPv6 addresses in Greenplum Database utility programs, or when you use numeric IP addresses instead of hostnames in any management utility, always enclose the IP address in brackets. When specifying an IP address at the command line, the best practice is to escape any brackets or enclose them in single quotes. For example, use either:

```
\[2620:0:170:610::11\]
```

Or:

```
'[2620:0:170:610::11]'
```

Running Backend Server Programs

Greenplum Database has modified certain PostgreSQL backend server programs to handle the parallelism and distribution of a Greenplum Database system. You access these programs only through the Greenplum Database management tools and utilities. *Do not run these programs directly.*

The following table identifies certain PostgreSQL backend server programs and the Greenplum Database utility command to run instead.

Table 63: Greenplum Database Backend Server Programs

PostgreSQL Program Name	Description	Use Instead
`initdb`	This program is called by `gpinitsystem` when initializing a Greenplum Database array. It is used internally to create the individual segment instances and the master instance.	*`gpinitsystem`*
`ipcclean`	Not used in Greenplum Database	N/A
`pg_basebackup`	This program makes a binary copy of a single database instance. Greenplum Database uses it for tasks such as creating a standby master instance, or recovering a mirror segment when a full copy is needed. Do not use this utility to back up Greenplum Database segment instances because it does not produce MPP-consistent backups.	*`gpinitstandby`*, *`gprecoverseg`*
`pg_controldata`	Not used in Greenplum Database	*`gpstate`*

PostgreSQL Program Name	Description	Use Instead
`pg_ctl`	This program is called by `gpstart` and `gpstop` when starting or stopping a Greenplum Database array. It is used internally to stop and start the individual segment instances and the master instance in parallel and with the correct options.	*`gpstart`*, *`gpstop`*
`pg_resetxlog`	DO NOT USE **Warning:** This program might cause data loss or cause data to become unavailable. If this program is used, the Pivotal Greenplum Database cluster is not supported. The cluster must be reinitialized and restored by the customer.	N/A
`postgres`	The `postgres` executable is the actual PostgreSQL server process that processes queries.	The main `postgres` process (postmaster) creates other `postgres` subprocesses and `postgres` session as needed to handle client connections.
`postmaster`	`postmaster` starts the `postgres` database server listener process that accepts client connections. In Greenplum Database, a `postgres` database listener process runs on the Greenplum master Instance and on each Segment Instance.	In Greenplum Database, you use *`gpstart`* and *`gpstop`* to start all postmasters (`postgres` processes) in the system at once in the correct order and with the correct options.

Utility Reference

The command-line utilities provided with Greenplum Database.

Greenplum Database uses the standard PostgreSQL client and server programs and provides additional management utilities for administering a distributed Greenplum Database DBMS.

Several utilities are installed when you install the Greenplum Database server. These utilities reside in `$GPHOME/bin`. Other utilities must be downloaded from Pivotal Network and installed separately. These include:

- The *Pivotal Greenplum Backup and Restore* utilities.
- The *Pivotal Greenplum gpcopy* utility.
- The *Pivotal Greenplum-Kafka Integration* utilities.
- The *Pivotal Greenplum Streaming Server* utilities.

Additionally, the Pivotal *Greenplum Client and Loader Tools* is a separate download from Pivotal Network that includes selected utilities from the Greenplum Database server installation that you can install on a client system.

Greenplum Database provides the following utility programs. Superscripts identify those utilities that require separate downloads, as well as those utilities that are also installed with the Client and Loader Tools Packages. (See the Note following the table.) All utilities are installed when you install the Greenplum Database server, unless specifically identified by a superscript.

Note:

[1] The utility program can be obtained from the *Greenplum Backup and Restore* tile on *Pivotal Network*.

[2] The utility program can be obtained from the *Greenplum Data Copy Utility* tile on *Pivotal Network*.

[3] The utility program is also installed with the *Greenplum Client and Loader Tools Package*s for Linux and Windows. You can obtain these packages from the Greenplum Database *Greenplum Clients* filegroup on *Pivotal Network*.

[4]The utility program is also installed with the *Greenplum Client and Loader Tools Package* for Linux. You can obtain the most up-to-date version of the *Greenplum Streaming Server* and *Greenplum-Kafka Integration* from *Pivotal Network*.

analyzedb

A utility that performs ANALYZE operations on tables incrementally and concurrently. For append optimized tables, `analyzedb` updates statistics only if the statistics are not current.

Synopsis

```
analyzedb -d dbname
   { -s schema  |
   { -t schema.table
     [ -i col1[, col2, ...] |
       -x col1[, col2, ...] ] } |
     { -f | --file} config-file }
   [ -l | --list ]
   [ --gen_profile_only ]
   [ -p parallel-level ]
   [ --full ]
   [ --skip_root_stats ]
   [ -v | --verbose ]
   [ --debug ]
```

```
   [ -a ]

analyzedb { --clean_last | --clean_all }
analyzedb --version
analyzedb { -? | -h | --help }
```

Description

The `analyzedb` utility updates statistics on table data for the specified tables in a Greenplum database incrementally and concurrently.

While performing *ANALYZE* operations, `analyzedb` creates a snapshot of the table metadata and stores it on disk on the master host. An ANALYZE operation is performed only if the table has been modified. If a table or partition has not been modified since the last time it was analyzed, `analyzedb` automatically skips the table or partition because it already contains up-to-date statistics.

- For append optimized tables, `analyzedb` updates statistics incrementally, if the statistics are not current. For example, if table data is changed after statistics were collected for the table. If there are no statistics for the table, statistics are collected.
- For heap tables, statistics are always updated.

Specify the `--full` option to update append-optimized table statistics even if the table statistics are current.

By default, `analyzedb` creates a maximum of 5 concurrent sessions to analyze tables in parallel. For each session, `analyzedb` issues an ANALYZE command to the database and specifies different table names. The `-p` option controls the maximum number of concurrent sessions.

Partitioned Append-Optimized Tables

For a partitioned, append-optimized table, `analyzedb` checks the partitioned table root partition and leaf partitions. If needed, the utility updates statistics for non-current partitions and the root partition.

The root partition statistics is required by GPORCA. The `analyzedb` utility collects statistics on the root partition of a partitioned table if the statistics do not exist. If any of the leaf partitions have stale statistics, `analyzedb` also refreshes the root partition statistics. The cost of refreshing the root level statistics is comparable to analyzing one leaf partition.

Notes

The `analyzedb` utility updates append optimized table statistics if the table has been modified by DML or DDL commands, including `INSERT`, `DELETE`, `UPDATE`, `CREATE TABLE`, `ALTER TABLE` and `TRUNCATE`. The utility determines if a table has been modified by comparing catalog metadata of tables with the previous snapshot of metadata taken during a previous `analyzedb` operation. The snapshots of table metadata are stored as state files in the directory `db_analyze/<db_name>/<timestamp>` in the Greenplum Database master data directory.

The utility does not automatically remove old snapshot information. Over time, the snapshots can consume a large amount of disk space. To recover disk space, you can keep the most recent snapshot information and remove the older snapshots. You can also specify the `--clean_last` or `--clean_all` option to remove state files generated by `analyzedb`.

If you do not specify a table, set of tables, or schema, the `analyzedb` utility collects the statistics as needed on all system catalog tables and user-defined tables in the database.

External tables are not affected by `analyzedb`.

Table names that contain spaces are not supported.

Running the ANALYZE command on a table, not using the `analyzedb` utility, does not update the table metadata that the `analyzedb` utility uses to determine whether table statistics are up to date.

Options

`--clean_last`

Remove the state files generated by last `analyzedb` operation. All other options except `-d` are ignored.

`--clean_all`

Remove all the state files generated by `analyzedb`. All other options except `-d` are ignored.

`-d` *`dbname`*

Specifies the name of the database that contains the tables to be analyzed. If this option is not specified, the database name is read from the environment variable `PGDATABASE`. If `PGDATABASE` is not set, the user name specified for the connection is used.

`--debug`

If specified, sets the logging level to debug. During command execution, debug level information is written to the log file and to the command line. The information includes the commands executed by the utility and the duration of each `ANALYZE` operation.

`-f` *`config-file`* | `--file` *`config-file`*

Text file that contains a list of tables to be analyzed. A relative file path from current directory can be specified.

The file lists one table per line. Table names must be qualified with a schema name. Optionally, a list of columns can be specified using the `-i` or `-x`. No other options are allowed in the file. Other options such as `--full` must be specified on the command line.

Only one of the options can be used to specify the files to be analyzed: `-f` or `--file`, `-t` , or `-s`.

When performing `ANALYZE` operations on multiple tables, `analyzedb` creates concurrent sessions to analyze tables in parallel. The `-p` option controls the maximum number of concurrent sessions.

In the following example, the first line performs an `ANALYZE` operation on the table `public.nation`, the second line performs an `ANALYZE` operation only on the columns `l_shipdate` and `l_receiptdate` in the table `public.lineitem`.

```
public.nation
public.lineitem -i l_shipdate, l_receiptdate
```

`--full`

Perform an `ANALYZE` operation on all the specified tables. The operation is performed even if the statistics are up to date.

`--gen_profile_only`

Update the `analyzedb` snapshot of table statistics information without performing any `ANALYZE` operations. If other options specify tables or a schema, the utility updates the snapshot information only for the specified tables.

Specify this option if the `ANALYZE` command was run on database tables and you want to update the `analyzedb` snapshot for the tables.

`-i` *`col1, col2, ...`*

Optional. Must be specified with the `-t` option. For the table specified with the `-t` option, collect statistics only for the specified columns.

Only `-i`, or `-x` can be specified. Both options cannot be specified.

`-l | --list`

Lists the tables that would have been analyzed with the specified options. The `ANALYZE` operations are not performed.

-p *parallel-level*

The number of tables that are analyzed in parallel. *parallel level* can be an integer between 1 and 10, inclusive. Default value is 5.

--skip_root_stats

Note: This option is deprecated and will be removed in a future release.

If the option is specified, a warning is issued stating that the option will be ignored.

GPORCA uses root partition statistics. For information about how statistics are collected for partitioned tables, see *ANALYZE*.

-s *schema*

Specify a schema to analyze. All tables in the schema will be analyzed. Only a single schema name can be specified on the command line.

Only one of the options can be used to specify the files to be analyzed: `-f` or `--file`, `-t` , or `-s`.

-t *schema.table*

Collect statistics only on *schema.table*. The table name must be qualified with a schema name. Only a single table name can be specified on the command line. You can specify the `-f` option to specify multiple tables in a file or the `-s` option to specify all the tables in a schema.

Only one of these options can be used to specify the files to be analyzed: `-f` or `--file`, `-t` , or `-s`.

-x *col1, col2, ...*

Optional. Must be specified with the `-t` option. For the table specified with the `-t` option, exclude statistics collection for the specified columns. Statistics are collected only on the columns that are not listed.

Only `-i`, or `-x` can be specified. Both options cannot be specified.

-a

Quiet mode. Do not prompt for user confirmation.

-h | -? | --help

Displays the online help.

-v | --verbose

If specified, sets the logging level to verbose to write additional information the log file and to the command line during command execution. The information includes a list of all the tables to be analyzed (including child leaf partitions of partitioned tables). Output also includes the duration of each ANALYZE operation.

--version

Displays the version of this utility.

Examples

An example that collects statistics only on a set of table columns. In the database `mytest`, collect statistics on the columns `shipdate` and `receiptdate` in the table `public.orders`:

```
analyzedb -d mytest -t public.orders -i shipdate, receiptdate
```

An example that collects statistics on a table and exclude a set of columns. In the database `mytest`, collect statistics on the table `public.foo`, and do not collect statistics on the columns `bar` and `test2`.

```
analyzedb -d mytest -t public.foo -x bar, test2
```

An example that specifies a file that contains a list of tables. This command collect statistics on the tables listed in the file `analyze-tables` in the database named `mytest`.

```
analyzedb -d mytest -f analyze-tables
```

If you do not specify a table, set of tables, or schema, the `analyzedb` utility collects the statistics as needed on all catalog tables and user-defined tables in the specified database. This command refreshes table statistics on the system catalog tables and user-defined tables in the database `mytest`.

```
analyzedb -d mytest
```

You can create a PL/Python function to run the `analyzedb` utility as a Greenplum Database function. This example `CREATE FUNCTION` command creates a user defined PL/Python function that runs the `analyzedb` utility and displays output on the command line. Specify `analyzedb` options as the function parameter.

```
CREATE OR REPLACE FUNCTION analyzedb(params TEXT)
  RETURNS VOID AS
$BODY$
    import subprocess
    cmd = ['analyzedb', '-a' ] + params.split()
    p = subprocess.Popen(cmd, stdout=subprocess.PIPE,
 stderr=subprocess.STDOUT)

    # verbose output of process
    for line in iter(p.stdout.readline, ''):
        plpy.info(line);

    p.wait()
$BODY$
LANGUAGE plpythonu VOLATILE;
```

When this `SELECT` command is run by the gpadmin user, the `analyzedb` utility performs an analyze operation on the table `public.mytable` that is in the database `mytest`.

```
SELECT analyzedb('-d mytest -t public.mytable') ;
```

> **Note:** To create a PL/Python function, the PL/Python procedural language must be registered as a language in the database. For example, this `CREATE LANGUAGE` command run as gpadmin registers PL/Python as an untrusted language:
>
> ```
> CREATE LANGUAGE plpythonu;
> ```

See Also

`ANALYZE` in the *Greenplum Database Reference Guide*

clusterdb

Reclusters tables that were previously clustered with `CLUSTER`.

Synopsis

```
clusterdb [connection-option ...] [--verbose | -v] [--table | -t table] [[--
dbname | -d] dbname]

clusterdb [connection-option ...] [--all | -a] [--verbose | -v]
```

```
clusterdb -? | --help

clusterdb -V | --version
```

Description

To cluster a table means to physically reorder a table on disk according to an index so that index scan operations can access data on disk in a somewhat sequential order, thereby improving index seek performance for queries that use that index.

The `clusterdb` utility will find any tables in a database that have previously been clustered with the `CLUSTER` SQL command, and clusters them again on the same index that was last used. Tables that have never been clustered are not affected.

`clusterdb` is a wrapper around the SQL command `CLUSTER`. Although clustering a table in this way is supported in Greenplum Database, it is not recommended because the `CLUSTER` operation itself is extremely slow.

If you do need to order a table in this way to improve your query performance, use a `CREATE TABLE AS` statement to reorder the table on disk rather than using `CLUSTER`. If you do 'cluster' a table in this way, then `clusterdb` would not be relevant.

Options

`-a | --all`

Cluster all databases.

`[-d] dbname | [--dbname=]dbname`

Specifies the name of the database to be clustered. If this is not specified, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

`-e | --echo`

Echo the commands that `clusterdb` generates and sends to the server.

`-q | --quiet`

Do not display a response.

`-t table | --table=table`

Cluster the named table only. Multiple tables can be clustered by writing multiple `-t` switches.

`-v | --verbose`

Print detailed information during processing.

`-V | --version`

Print the `clusterdb` version and exit.

`-? | --help`

Show help about `clusterdb` command line arguments, and exit.

Connection Options

`-h host | --host=host`

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

`-p port | --port=port`

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

`-U username | --username=username`

The database role name to connect as. If not specified, reads from the environment variable PGUSER or defaults to the current system role name.

-w | --no-password

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

-W | --password

Force a password prompt.

--maintenance-db=*dbname*

Specifies the name of the database to connect to discover what other databases should be clustered. If not specified, the `postgres` database will be used, and if that does not exist, `template1` will be used.

Examples

To cluster the database `test`:

```
clusterdb test
```

To cluster a single table `foo` in a database named `xyzzy`:

```
clusterdb --table foo xyzzyb
```

See Also

`CLUSTER` in the *Greenplum Database Reference Guide*

createdb

Creates a new database.

Synopsis

```
createdb [connection-option ...] [option ...] [dbname ['description']]

createdb -? | --help

createdb -V | --version
```

Description

`createdb` creates a new database in a Greenplum Database system.

Normally, the database user who executes this command becomes the owner of the new database. However, a different owner can be specified via the `-O` option, if the executing user has appropriate privileges.

`createdb` is a wrapper around the SQL command *`CREATE DATABASE`* .

Options

dbname

The name of the database to be created. The name must be unique among all other databases in the Greenplum system. If not specified, reads from the environment variable PGDATABASE, then PGUSER or defaults to the current system user.

description

A comment to be associated with the newly created database. Descriptions containing white space must be enclosed in quotes.

-D *tablespace* | --tablespace=*tablespace*

Specifies the default tablespace for the database. (This name is processed as a double-quoted identifier.)

-e echo

Echo the commands that `createdb` generates and sends to the server.

-E *encoding* | --encoding *encoding*

Character set encoding to use in the new database. Specify a string constant (such as `'UTF8'`), an integer encoding number, or `DEFAULT` to use the default encoding. See the Greenplum Database Reference Guide for information about supported character sets.

-l *locale* | --locale *locale*

Specifies the locale to be used in this database. This is equivalent to specifying both `--lc-collate` and `--lc-ctype`.

--lc-collate *locale*

Specifies the `LC_COLLATE` setting to be used in this database.

--lc-ctype *locale*

Specifies the LC_CTYPE setting to be used in this database.

--maintenance-db=*dbname*

Specifies the name of the database to connect to when creating the new database. If not specified, the `postgres` database will be used; if that does not exist (or if it is the name of the new database being created), `template1` will be used.

-O *owner* | --owner=*owner*

The name of the database user who will own the new database. Defaults to the user executing this command. (This name is processed as a double-quoted identifier.)

-T *template* | --template=*template*

The name of the template from which to create the new database. Defaults to `template1`. (This name is processed as a double-quoted identifier.)

-V | --version

Print the `createdb` version and exit.

-? | --help

Show help about `createdb` command line arguments, and exit.

The options `-D`, `-l`, `-E`, `-O`, and `-T` correspond to options of the underlying SQL command `CREATE DATABASE`; see `CREATE DATABASE` in the *Greenplum Database Reference Guide* for more information about them.

Connection Options

-h *host* | --host=*host*

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

-p *port* | --port=*port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username=*username*

The database role name to connect as. If not specified, reads from the environment variable PGUSER or defaults to the current system role name.

-w | --no-password

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a .pgpass file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

-W | --password

Force a password prompt.

Examples

To create the database test using the default options:

```
createdb test
```

To create the database demo using the Greenplum master on host gpmaster, port 54321, using the LATIN1 encoding scheme:

```
createdb -p 54321 -h gpmaster -E LATIN1 demo
```

See Also

CREATE DATABASE in the *Greenplum Database Reference Guide*

createlang

Defines a new procedural language for a database.

Synopsis

```
createlang [connection_option ...] [-e] langname [[-d] dbname]

createlang [connection-option ...] -l dbname

createlang -? | --help

createlang -V | --version
```

Description

The createlang utility adds a new procedural language to a database. createlang is a wrapper around the SQL command *CREATE EXTENSION*.

> **Note:** createlang is deprecated and may be removed in a future release. From Greenplum Database 6.x, using createlang to add a procedural language package generates an error. Use the CREATE EXTENSION command instead.

The procedural language packages included in the standard Greenplum Database distribution are:

- PL/pgSQL
- PL/Perl
- PL/Python

The PL/pgSQL language is registered in all databases by default.

Greenplum Database also has language handlers for `PL/Java` and `PL/R`, but those languages are not pre-installed with Greenplum Database. See the *Greenplum PL/Java Language Extension* and *Greenplum PL/R Language Extension* sections in the documentation for more information.

Options

`langname`

Specifies the name of the procedural language to be installed. (This name is lower-cased.)

`[-d] dbname | [--dbname=]dbname`

Specifies the database to which the language should be added. The default is to use the `PGDATABASE` environment variable setting, or the same name as the current system user.

`-e | --echo`

Echo the commands that `createlang` generates and sends to the server.

`-l dbname | --list dbname`

Show a list of already installed languages in the target database.

`-V | --version`

Print the `createlang` version and exit.

`-? | --help`

Show help about `createlang` command line arguments, and exit.

Connection Options

`-h host | --host=host`

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

`-p port | --port=port`

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to `5432`.

`-U username | --username=username`

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

`-w | --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W | --password`

Force a password prompt.

Examples

To install the language `plperl` into the database `mytestdb`:

```
createlang plperl mytestdb
```

See Also

createuser

Creates a new database role.

Synopsis

```
createuser [connection-option ...] [role_attribute ...] [-e] role_name

createuser -? | --help

createuser -V | --version
```

Description

`createuser` creates a new Greenplum Database role. You must be a superuser or have the `CREATEROLE` privilege to create new roles. You must connect to the database as a superuser to create new superusers.

Superusers can bypass all access permission checks within the database, so superuser privileges should not be granted lightly.

`createuser` is a wrapper around the SQL command `CREATE ROLE`.

Options

role_name

The name of the role to be created. This name must be different from all existing roles in this Greenplum Database installation.

-c *number* | --connection-limit=*number*

Set a maximum number of connections for the new role. The default is to set no limit.

-d | --createdb

The new role will be allowed to create databases.

-D | --no-createdb

The new role will not be allowed to create databases. This is the default.

-e | --echo

Echo the commands that `createuser` generates and sends to the server.

-E | --encrypted

Encrypts the role's password stored in the database. If not specified, the default password behavior is used.

-i | --inherit

The new role will automatically inherit privileges of roles it is a member of. This is the default.

-I | --no-inherit

The new role will not automatically inherit privileges of roles it is a member of.

--interactive

Prompt for the user name if none is specified on the command line, and also prompt for whichever of the options `-d`/`-D`, `-r`/`-R`, `-s`/`-S` is not specified on the command line.

-l | --login

The new role will be allowed to log in to Greenplum Database. This is the default.

-L | --no-login

The new role will not be allowed to log in (a group-level role).

-N | --unencrypted

Does not encrypt the role's password stored in the database. If not specified, the default password behavior is used.

-P | --pwprompt

If given, `createuser` will issue a prompt for the password of the new role. This is not necessary if you do not plan on using password authentication.

-r | --createrole

The new role will be allowed to create new roles (`CREATEROLE` privilege).

-R | --no-createrole

The new role will not be allowed to create new roles. This is the default.

-s | --superuser

The new role will be a superuser.

-S | --no-superuser

The new role will not be a superuser. This is the default.

-V | --version

Print the `createuser` version and exit.

--replication

The new user will have the `REPLICATION` privilege, which is described more fully in the documentation for *CREATE ROLE*.

--no-replication

The new user will not have the `REPLICATION` privilege, which is described more fully in the documentation for *CREATE ROLE*.

-? | --help

Show help about `createuser` command line arguments, and exit.

Connection Options

-h *host* | --host=*host*

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

-p *port* | --port=*port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username=*username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-w | --no-password

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

-W | --password

Force a password prompt.

Examples

To create a role `joe` on the default database server:

```
$ createuser joe
```

To create a role `joe` on the default database server with prompting for some additional attributes:

```
$ createuser --interactive joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
CREATE ROLE
```

To create the same role `joe` using connection options, with attributes explicitly specified, and taking a look at the underlying command:

```
createuser -h masterhost -p 54321 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT
LOGIN;
CREATE ROLE
```

To create the role `joe` as a superuser, and assign password `admin123` immediately:

```
createuser -P -s -e joe
Enter password for new role: admin123
Enter it again: admin123
CREATE ROLE joe PASSWORD 'admin123' SUPERUSER CREATEDB
CREATEROLE INHERIT LOGIN;
CREATE ROLE
```

In the above example, the new password is not actually echoed when typed, but we show what was typed for clarity. However the password will appear in the echoed command, as illustrated if the `-e` option is used.

See Also

`CREATE ROLE` in the *Greenplum Database Reference Guide*

dropdb

Removes a database.

Synopsis

```
dropdb [connection-option ...] [-e] [-i] dbname

dropdb -? | --help

dropdb -V | --version
```

Description

`dropdb` destroys an existing database. The user who executes this command must be a superuser or the owner of the database being dropped.

`dropdb` is a wrapper around the SQL command `DROP DATABASE`. See the *Greenplum Database Reference Guide* for information about `DROP DATABASE`.

Options

dbname

The name of the database to be removed.

-e | --echo

Echo the commands that `dropdb` generates and sends to the server.

`-i | --interactive`

Issues a verification prompt before doing anything destructive.

`-V | --version`

Print the `dropdb` version and exit.

`--if-exists`

Do not throw an error if the database does not exist. A notice is issued in this case.

`-? | --help`

Show help about `dropdb` command line arguments, and exit.

Connection Options

`-h *host* | --host=*host*`

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

`-p *port* | --port=*port*`

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

`-U *username* | --username=*username*`

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

`-w | --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W | --password`

Force a password prompt.

`--maintenance-db=*dbname*`

Specifies the name of the database to connect to in order to drop the target database. If not specified, the `postgres` database will be used; if that does not exist (or if it is the name of the database being dropped), `template1` will be used.

Examples

To destroy the database named `demo` using default connection parameters:

```
dropdb demo
```

To destroy the database named `demo` using connection options, with verification, and a peek at the underlying command:

```
dropdb -p 54321 -h masterhost -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE "demo"
DROP DATABASE
```

See Also

`DROP DATABASE` in the *Greenplum Database Reference Guide*

droplang

Removes a procedural language.

Synopsis

```
droplang [connection-option ...] [-e] langname [[-d] dbname]

droplang [connection-option ...] [-e] -l dbname

droplang -? | --help

droplang -V | --version
```

Description

`droplang` removes an existing procedural language from a database.

`droplang` is a wrapper for the SQL command *`DROP EXTENSION`*.

> **Note:** `droplang` is deprecated and may be removed in a future release. From Greenplum Database 6.x using `droplang` to remove a procedural language package generates an error. Use the `DROP EXTENSION` command instead.

Options

`langname`

Specifies the name of the procedural language to be removed. (This name is lower-cased.)

`[-d] dbname | [--dbname=]dbname`

Specifies from which database the language should be removed. The default is to use the `PGDATABASE` environment variable setting, or the same name as the current system user.

`-e | --echo`

Echo the commands that `droplang` generates and sends to the server.

`-l | --list`

Show a list of already installed languages in the target database.

`-V | --version`

Print the `droplang` version and exit.

`-? | --help`

Show help about `droplang` command line arguments, and exit.

Connection Options

`-h host | --host=host`

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

`-p port | --port=port`

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

`-U username | --username=username`

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

`-w | --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W | --password`

Force a password prompt.

Examples

To remove the language `pltcl` from the `mydatabase` database:

```
droplang pltcl mydatabase
```

See Also

dropuser

Removes a database role.

Synopsis

```
dropuser [connection-option ...] [-e] [-i] role_name

dropuser -? | --help

dropuser -V | --version
```

Description

`dropuser` removes an existing role from Greenplum Database. Only superusers and users with the `CREATEROLE` privilege can remove roles. To remove a superuser role, you must yourself be a superuser.

`dropuser` is a wrapper around the SQL command `DROP ROLE`.

Options

`role_name`

The name of the role to be removed. You will be prompted for a name if not specified on the command line and the `-i`/`--interactive` option is used.

`-e | --echo`

Echo the commands that `dropuser` generates and sends to the server.

`-i | --interactive`

Prompt for confirmation before actually removing the role, and prompt for the role name if none is specified on the command line.

`--if-exists`

Do not throw an error if the user does not exist. A notice is issued in this case.

`-V | --version`

Print the `dropuser` version and exit.

`-? | --help`

Show help about `dropuser` command line arguments, and exit.

Connection Options

`-h *host* | --host=*host*`

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable PGHOST or defaults to localhost.

-p *port* | --port=*port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable PGPORT or defaults to 5432.

-U *username* | --username=*username*

The database role name to connect as. If not specified, reads from the environment variable PGUSER or defaults to the current system role name.

-w | --no-password

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

-W | --password

Force a password prompt.

Examples

To remove the role `joe` using default connection options:

```
dropuser joe
DROP ROLE
```

To remove the role *joe* using connection options, with verification, and a peek at the underlying command:

```
dropuser -p 54321 -h masterhost -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE "joe"
DROP ROLE
```

See Also

`DROP ROLE` in the *Greenplum Database Reference Guide*

gpactivatestandby

Activates a standby master host and makes it the active master for the Greenplum Database system.

Synopsis

```
gpactivatestandby [-d standby_master_datadir] [-f] [-a] [-q]
    [-l logfile_directory]

gpactivatestandby -v

gpactivatestandby -? | -h | --help
```

Description

The `gpactivatestandby` utility activates a backup, standby master host and brings it into operation as the active master instance for a Greenplum Database system. The activated standby master effectively becomes the Greenplum Database master, accepting client connections on the master port.

When you initialize a standby master, the default is to use the same port as the active master. For information about the master port for the standby master, see *gpinitstandby*.

You must run this utility from the master host you are activating, not the failed master host you are disabling. Running this utility assumes you have a standby master host configured for the system (see *gpinitstandby*).

The utility will perform the following steps:

- Stops the synchronization process (walreceiver) on the standby master
- Updates the system catalog tables of the standby master using the logs
- Activates the standby master to be the new active master for the system
- Restarts the Greenplum Database system with the new master host

A backup, standby Greenplum master host serves as a 'warm standby' in the event of the primary Greenplum master host becoming non-operational. The standby master is kept up to date by transaction log replication processes (the walsender and walreceiver), which run on the primary master and standby master hosts and keep the data between the primary and standby master hosts synchronized.

If the primary master fails, the log replication process is shutdown, and the standby master can be activated in its place by using the gpactivatestandby utility. Upon activation of the standby master, the replicated logs are used to reconstruct the state of the Greenplum master host at the time of the last successfully committed transaction.

In order to use gpactivatestandby to activate a new primary master host, the master host that was previously serving as the primary master cannot be running. The utility checks for a postmaster.pid file in the data directory of the disabled master host, and if it finds it there, it will assume the old master host is still active. In some cases, you may need to remove the postmaster.pid file from the disabled master host data directory before running gpactivatestandby (for example, if the disabled master host process was terminated unexpectedly).

After activating a standby master, run ANALYZE to update the database query statistics. For example:

```
psql dbname -c 'ANALYZE;'
```

After you activate the standby master as the primary master, the Greenplum Database system no longer has a standby master configured. You might want to specify another host to be the new standby with the *gpinitstandby* utility.

Options

-a (do not prompt)

Do not prompt the user for confirmation.

-d *standby_master_datadir*

The absolute path of the data directory for the master host you are activating.

If this option is not specified, gpactivatestandby uses the value of the MASTER_DATA_DIRECTORY environment variable setting on the master host you are activating. If this option is specified, it overrides any setting of MASTER_DATA_DIRECTORY.

If a directory cannot be determined, the utility returns an error.

-f (force activation)

Use this option to force activation of the backup master host. Use this option only if you are sure that the standby and primary master hosts are consistent.

-l *logfile_directory*

The directory to write the log file. Defaults to ~/gpAdminLogs.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

-? | -h | --help (help)

Displays the online help.

Example

Activate the standby master host and make it the active master instance for a Greenplum Database system (run from backup master host you are activating):

```
gpactivatestandby -d /gpdata
```

See Also

gpinitsystem, *gpinitstandby*

gpaddmirrors

Adds mirror segments to a Greenplum Database system that was initially configured without mirroring.

Synopsis

```
gpaddmirrors [-p port_offset] [-m datadir_config_file [-a]] [-s]
   [-d master_data_directory] [-B parallel_processes] [-l logfile_directory]
   [-v]

gpaddmirrors -i mirror_config_file [-a] [-d master_data_directory]
   [-B parallel_processes] [-l logfile_directory] [-v]

gpaddmirrors -o output_sample_mirror_config [-s] [-m datadir_config_file]

gpaddmirrors -?

gpaddmirrors --version
```

Description

The `gpaddmirrors` utility configures mirror segment instances for an existing Greenplum Database system that was initially configured with primary segment instances only. The utility will create the mirror instances and begin the online replication process between the primary and mirror segment instances. Once all mirrors are synchronized with their primaries, your Greenplum Database system is fully data redundant.

Important: During the online replication process, Greenplum Database should be in a quiescent state, workloads and other queries should not be running.

By default, the utility will prompt you for the file system location(s) where it will create the mirror segment data directories. If you do not want to be prompted, you can pass in a file containing the file system locations using the `-m` option.

The mirror locations and ports must be different than your primary segment data locations and ports.

The utility creates a unique data directory for each mirror segment instance in the specified location using the predefined naming convention. There must be the same number of file system locations declared for mirror segment instances as for primary segment instances. It is OK to specify the same directory name

multiple times if you want your mirror data directories created in the same location, or you can enter a different data location for each mirror. Enter the absolute path. For example:

```
Enter mirror segment data directory location 1 of 2 > /gpdb/mirror
Enter mirror segment data directory location 2 of 2 > /gpdb/mirror
```

OR

```
Enter mirror segment data directory location 1 of 2 > /gpdb/m1
Enter mirror segment data directory location 2 of 2 > /gpdb/m2
```

Alternatively, you can run the `gpaddmirrors` utility and supply a detailed configuration file using the `-i` option. This is useful if you want your mirror segments on a completely different set of hosts than your primary segments. The format of the mirror configuration file is:

```
<contentID>|<address>|<port>|<data_dir>
```

Where `<contentID>` is the segment instance content ID, `<address>` is the host name or IP address of the segment host, `<port>` is the communication port, and `<data_dir>` is the segment instance data directory.

For example:

```
0|sdw1-1|60000|/gpdata/m1/gp0
1|sdw1-1|60001|/gpdata/m2/gp1
```

The `gp_segment_configuration` system catalog table can help you determine your current primary segment configuration so that you can plan your mirror segment configuration. For example, run the following query:

```
=# SELECT dbid, content, address as host_address, port, datadir
   FROM gp_segment_configuration
   ORDER BY dbid;
```

If you are creating mirrors on alternate mirror hosts, the new mirror segment hosts must be pre-installed with the Greenplum Database software and configured exactly the same as the existing primary segment hosts.

You must make sure that the user who runs `gpaddmirrors` (the `gpadmin` user) has permissions to write to the data directory locations specified. You may want to create these directories on the segment hosts and `chown` them to the appropriate user before running `gpaddmirrors`.

> **Note:** This utility uses secure shell (SSH) connections between systems to perform its tasks. In large Greenplum Database deployments, cloud deployments, or deployments with a large number of segments per host, this utility may exceed the host's maximum threshold for unauthenticated connections. Consider updating the SSH `MaxStartups` configuration parameter to increase this threshold. For more information about SSH configuration options, refer to the SSH documentation for your Linux distribution.

Options

`-a (do not prompt)`

Run in quiet mode - do not prompt for information. Must supply a configuration file with either `-m` or `-i` if this option is used.

`-B` *`parallel_processes`*

The number of mirror setup processes to start in parallel. If not specified, the utility will start up to 10 parallel processes depending on how many mirror segment instances it needs to set up.

-d *master_data_directory*

The master data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-i *mirror_config_file*

A configuration file containing one line for each mirror segment you want to create. You must have one mirror segment instance listed for each primary segment in the system. The format of this file is as follows (as per attributes in the *gp_segment_configuration* catalog table):

```
<contentID>|<address>|<port>|<data_dir>
```

Where `<contentID>` is the segment instance content ID, `<address>` is the hostname or IP address of the segment host, `<port>` is the communication port, and `<data_dir>` is the segment instance data directory. For information about using a hostname or IP address, see *Specifying Hosts using Hostnames or IP Addresses*. Also, see *Using Host Systems with Multiple NICs*.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-m *datadir_config_file*

A configuration file containing a list of file system locations where the mirror data directories will be created. If not supplied, the utility prompts you for locations. Each line in the file specifies a mirror data directory location. For example:

```
/gpdata/m1
/gpdata/m2
/gpdata/m3
/gpdata/m4
```

-o *output_sample_mirror_config*

If you are not sure how to lay out the mirror configuration file used by the `-i` option, you can run `gpaddmirrors` with this option to generate a sample mirror configuration file based on your primary segment configuration. The utility will prompt you for your mirror segment data directory locations (unless you provide these in a file using `-m`). You can then edit this file to change the host names to alternate mirror hosts if necessary.

-p *port_offset*

Optional. This number is used to calculate the database ports used for mirror segments. The default offset is 1000. Mirror port assignments are calculated as follows:

```
primary_port + offset = mirror_database_port
```

For example, if a primary segment has port 50001, then its mirror will use a database port of 51001, by default.

-s (spread mirrors)

Spreads the mirror segments across the available hosts. The default is to group a set of mirror segments together on an alternate host from their primary segment set. Mirror spreading will place each mirror on a different host within the Greenplum Database array. Spreading is only allowed if there is a sufficient number of hosts in the array (number of hosts is greater than the number of segment instances per host).

-v (verbose)

Sets logging output to verbose.

--version (show utility version)

Displays the version of this utility.

-? (help)

Displays the online help.

Specifying Hosts using Hostnames or IP Addresses

When specifying a mirroring configuration using the `gpaddmirrors` option `-i`, you can specify either a hostname or an IP address for the <address> value.

- If you specify a hostname, the resolution of the hostname to an IP address should be done locally for security. For example, you should use entries in a local `/etc/hosts` file to map the hostname to an IP address. The resolution of a hostname to an IP address should not be performed by an external service such as a public DNS server. You must stop the Greenplum system before you change the mapping of a hostname to a different IP address.
- If you specify an IP address, the address should not be changed after the initial configuration. When segment mirroring is enabled, replication from the primary to the mirror segment will fail if the IP address changes from the configured value. For this reason, you should use a hostname when enabling mirroring using the `-i` option unless you have a specific requirement to use IP addresses.

When enabling a mirroring configuration that adds hosts to the Greenplum system, `gpaddmirrors` populates the *`gp_segment_configuration`* catalog table with the mirror segment instance information. Greenplum Database uses the *address* value of the `gp_segment_configuration` catalog table when looking up host systems for Greenplum interconnect (internal) communication between the master and segment instances and between segment instances, and for other internal communication.

Using Host Systems with Multiple NICs

If hosts systems are configured with multiple NICs, you can initialize a Greenplum Database system to use each NIC as a Greenplum host system. You must ensure that the host systems are configured with sufficient resources to support all the segment instances being added to the host. Also, if you enable segment mirroring, you must ensure that the Greenplum system configuration supports failover if a host system fails. For information about Greenplum Database mirroring schemes, see *Segment Mirroring Configurations*.

For example, this is a segment instance configuration for a simple Greenplum system. The segment host `gp6m` is configured with two NICs, `gp6m-1` and `gp6m-2`, where the Greenplum Database system uses `gp6m-1` for the master segment and `gp6m-2` for segment instances.

```
select content, role, port, hostname, address from
 gp_segment_configuration ;

 content | role | port  | hostname | address
---------+------+-------+----------+----------
      -1 | p    |  5432 | gp6m     | gp6m-1
       0 | p    | 40000 | gp6m     | gp6m-2
       0 | m    | 50000 | gp6s     | gp6s
       1 | p    | 40000 | gp6s     | gp6s
       1 | m    | 50000 | gp6m     | gp6m-2
(5 rows)
```

Examples

Add mirroring to an existing Greenplum Database system using the same set of hosts as your primary data. Calculate the mirror database ports by adding 100 to the current primary segment port numbers:

```
$ gpaddmirrors -p 100
```

Generate a sample mirror configuration file with the `-o` option to use with `gpaddmirrors -i`:

```
$ gpaddmirrors -o /home/gpadmin/sample_mirror_config
```

Add mirroring to an existing Greenplum Database system using a different set of hosts from your primary data:

```
$ gpaddmirrors -i mirror_config_file
```

Where `mirror_config_file` looks something like this:

```
0|sdw1-1|52001|/gpdata/m1/gp0
1|sdw1-2|52002|/gpdata/m2/gp1
2|sdw2-1|52001|/gpdata/m1/gp2
3|sdw2-2|52002|/gpdata/m2/gp3
```

See Also

gpinitsystem, *gpinitstandby*, *gpactivatestandby*

gpbackup_manager

Display information about existing backups, delete existing backups, or encrypt passwords for secure storage in plugin configuration files.

Synopsis

```
gpbackup_manager [command]
```

where *command* is:

```
delete-backup timestamp [--plugin-config config-file]
| display-report timestamp
| encrypt-password --plugin-config config-file
| list-backups
| help [command]
```

Commands

`delete-backup` *`timestamp`*

Deletes the backup set with the specified timestamp.

`display-report` *`timestamp`*

Displays the backup report for a specified timestamp.

`encrypt-password`

Encrypts plain-text passwords for storage in the DD Boost plugin configuration file.

`list-backups`

Displays a list of backups that have been taken. If the backup history file does not exist, the command exits with an error message. See *Table 64: Backup List Report* for a description of the columns in this list.

`help` *`command`*

Displays a help message for the specified command.

Options

`--plugin-config` *`config-file`*

The `delete-backup` command requires this option if the backup Is stored in s3 or a Data Domain system. The `encrypt-password` command requires this option.

`-h | --help`

Displays a help message for the `gpbackup_manager` command. For help on a specific `gpbackup_manager` command, enter `gpbackup_manager help` *`command`*. For example:

```
$ gpbackup_manager help encrypt-password
```

Description

The `gpbackup_manager` utility manages backup sets created using the *`gpbackup`* utility. You can list backups, display a report for a backup, and delete a backup. `gpbackup_manager` can also encrypt passwords to store in a DD Boost plugin configuration file.

Greenplum Database must be running to use the `gpbackup_manager` utility.

Backup history is saved on the Greenplum Database master host in the file `$MASTER_DATA_DIRECTORY/gpbackup_history.yaml`. If no backups have been created yet, or if the backup history has been deleted, `gpbackup_manager` commands that depend on the file will display an error message and exit. If the backup history contains invalid YAML syntax, a yaml error message is displayed.

Versions of `gpbackup` earlier than v1.13.0 did not save the backup duration in the backup history file. The `list-backups` command duration column is empty for these backups.

The `encrypt-password` command is used to encrypt Data Domain user passwords that are saved in a DD Boost plug-In configuration file. To use this option, the `pgcrypto` extension must be enabled in the Greenplum Database `postgres` database. See the Pivotal Greenplum Backup and Restore installation instructions for help installing `pgcrypto`.

The `encrypt-password` command prompts you to enter and then re-enter the password to be encrypted. To maintain password secrecy, characters entered are echoed as asterisks. If replication is enabled in the specified DD Boost configuration file, the command also prompts for a password for the remote Data Domain account. You must then copy the output of the command into the DD Boost configuration file.

The following table describes the contents of the columns in the list that is output by the `gpbackup_manager list-backups` command.

Table 64: Backup List Report

Column	Description
timestamp	Timestamp value (`YYYYMMDDHHMMSS`) that specifies the time the backup was taken.
date	Date the backup was taken.
database	Name of the database backed up (specified on the `gpbackup` command line with the `--dbname` option).

Column	Description
type	Which classes of data are included in the backup. Can be one of the following: **full** - contains all global and local metadata, and user data for the database. This kind of backup can be the base for an incremental backup. Depending on the `gpbackup` options specified, some objects could have been filtered from the backup. **incremental** – contains all global and local metadata, and user data changed since a previous **full** backup. **metadata-only** – contains only the global and local metadata for the database. Depending on the `gpbackup` options specified, some objects could have been filtered from the backup. **data-only** – contains only user data from the database. Depending on the `gpbackup` options specified, some objects could have been filtered from the backup.
object filtering	The object filtering options that were specified at least once on the `gpbackup` command line, or blank if no filtering operations were used. To see the object filtering details for a specific backup, run the `gpbackup_manager report` command for the backgit st **include-schema** – at least one `--include-schema` option was specified. **exclude-schema** – at least one `--exclude-schema` option was specified. **include-table** – at least one `--include-table` option was specified. **exclude-table** – at least one `--exclude-table` option was specified.
plugin	The name of the binary plugin file that was used to configure the backup destination, excluding path information.
duration	The amount of time (`hh:mm:ss` format) taken to complete the backup.
date deleted	The date the backup was deleted, or blank if the backup still exists.

Examples

1. Display a list of the existing backups.

```
gpadmin@mdw:$ gpbackup_manager list-backups
  timestamp        date                        database   type
 object filtering   plugin   duration   date deleted
  20190719092809   Fri Jul 19 2019 09:28:09   sales      full
 include-schema              01:49:38   Fri Jul 19 2019 09:30:34
  20190719092716   Fri Jul 19 2019 09:27:16   sales      full
 exclude-schema              01:38:45
  20190719092609   Fri Jul 19 2019 09:26:09   sales      data-only
                             01:07:22
  20190719092557   Fri Jul 19 2019 09:25:57   sales      metadata-only
                             00:00:19
  20190719092530   Fri Jul 19 2019 09:25:30   sales      full
                             01:50:27
```

2. Display the backup report for the backup with timestamp 20190612154608.

```
$ gpbackup_manager display-report 20190612154608

Greenplum Database Backup Report

Timestamp Key: 20190612154608
GPDB Version: 5.14.0+dev.8.gdb327b2a3f build
 commit:db327b2a3f6f2b0673229e9aa164812e3bb56263
gpbackup Version: 1.11.0
Database Name: sales
Command Line: gpbackup --dbname sales
Compression: gzip
Plugin Executable: None
Backup Section: All Sections
Object Filtering: None
Includes Statistics: No
Data File Format: Multiple Data Files Per Segment
Incremental: False
Start Time: 2019-06-12 15:46:08
End Time: 2019-06-12 15:46:53
Duration: 0:00:45

Backup Status: Success
Database Size: 3306 MB

Count of Database Objects in Backup:
Aggregates                   12
Casts                        4
Constraints                  0
Conversions                  0
Database GUCs                0
Extensions                   0
Functions                    0
Indexes                      0
Operator Classes             0
Operator Families            1
Operators                    0
Procedural Languages         1
Protocols                    1
Resource Groups              2
Resource Queues              6
Roles                        859
Rules                        0
Schemas                      185
Sequences                    207
Tables                       431
Tablespaces                  0
Text Search Configurations   0
Text Search Dictionaries     0
Text Search Parsers          0
Text Search Templates        0
Triggers                     0
Types                        2
Views                        0
```

3. Delete the local backup with timestamp 20190620145126.

```
$ gpbackup_manager delete-backup 20190620145126

Are you sure you want to delete-backup 20190620145126? (y/n)y
Deletion of 20190620145126 in progress.
```

```
Deletion of 20190620145126 complete.
```

4. Delete a backup stored on a Data Domain system. The DD Boost plugin configuration file must be specified with the `--plugin-config` option.

```
$ gpbackup_manager delete-backup 20190620160656 --plugin-config ~/
ddboost_config.yaml

Are you sure you want to delete-backup 20190620160656? (y/n)y
Deletion of 20190620160656 in progress.

Deletion of 20190620160656 done.
```

5. Encrypt a password. A DD Boost plugin configuration file must be specified with the `--plugin-config` option.

```
$ gpbackup_manager encrypt-password --plugin-config ~/
ddboost_rep_on_config.yaml

Please enter your password ******
Please verify your password ******
Please enter your remote password ******
Please verify your remote password ******

Please copy/paste these lines into the plugin config file:

password:
 "c30d04090302a0ff861b823d71b079d23801ac367a74a1a8c088ed53beb62b7e190b7110277ea5b51c8
password_encryption: "on"
remote_password:
 "c30d04090302c764fd06bfa1dade62d2380160a8f1e4d1ff0a4bb25a542fb1d31c7a19b98e9b2f00e7b
remote_password_encryption: "on"
```

See Also

`gprestore`, *Parallel Backup with gpbackup and gprestore* and *Using the S3 Storage Plugin with gpbackup and gprestore*

gpbackup

Create a Greenplum Database backup for use with the `gprestore` utility.

Synopsis

```
gpbackup --dbname database_name
   [--backup-dir directory]
   [--compression-level level]
   [--data-only]
   [--debug]
   [--exclude-schema schema_name [--exclude-schema schema_name ...]]
   [--exclude-table schema.table [--exclude-table schema.table ...]]
   [--exclude-schema-file file_name]
   [--exclude-table-file file_name]
   [--include-schema schema_name [--include-schema schema_name ...]]
   [--include-table schema.table [--include-table schema.table ...]]
   [--include-schema-file file_name]
   [--include-table-file file_name]
   [--incremental [--from-timestamp backup-timestamp]]
   [--jobs int]
   [--leaf-partition-data]
   [--metadata-only]
```

```
    [--no-compression]
    [--plugin-config config_file_location]
    [--quiet]
    [--single-data-file]
    [--verbose]
    [--version]
    [--with-stats]

gpbackup --help
```

Description

The gpbackup utility backs up the contents of a database into a collection of metadata files and data files that can be used to restore the database at a later time using gprestore. When you back up a database, you can specify table level and schema level filter options to back up specific tables. For example, you can combine schema level and table level options to back up all the tables in a schema except for a single table.

By default, gpbackup backs up objects in the specified database as well as global Greenplum Database system objects. You can optionally supply the --with-globals option with gprestore to restore global objects. See *Objects Included in a Backup or Restore* for additional information.

For materialized views, data is not backed up, only the materialized view definition is backed up.

gpbackup stores the object metadata files and DDL files for a backup in the Greenplum Database master data directory by default. Greenplum Database segments use the COPY ... ON SEGMENT command to store their data for backed-up tables in compressed CSV data files, located in each segment's data directory. See *Understanding Backup Files* for additional information.

You can add the --backup-dir option to copy all backup files from the Greenplum Database master and segment hosts to an absolute path for later use. Additional options are provided to filter the backup set in order to include or exclude specific tables.

You can create an incremental backup with the *--incremental* option. Incremental backups are efficient when the total amount of data in append-optimized tables or table partitions that changed is small compared to the data has not changed. See *Creating and Using Incremental Backups with gpbackup and gprestore* for information about incremental backups.

With the default --jobs option (1 job), each gpbackup operation uses a single transaction on the Greenplum Database master host. The COPY ... ON SEGMENT command performs the backup task in parallel on each segment host. The backup process acquires an ACCESS SHARE lock on each table that is backed up. During the table locking process, the database should be in a quiescent state.

When a back up operation completes, gpbackup returns a status code. See *Return Codes*.

The gpbackup utility cannot be run while gpexpand is initializing new segments. Backups created before the expansion cannot be restored with gprestore after the cluster expansion is completed.

gpbackup can send status email notifications after a back up operation completes. You specify when the utility sends the mail and the email recipients in a configuration file. See *Configuring Email Notifications*.

> **Note:** This utility uses secure shell (SSH) connections between systems to perform its tasks. In large Greenplum Database deployments, cloud deployments, or deployments with a large number of segments per host, this utility may exceed the host's maximum threshold for unauthenticated connections. Consider updating the SSH MaxStartups and MaxSessions configuration parameters to increase this threshold. For more information about SSH configuration options, refer to the SSH documentation for your Linux distribution.

Options

--dbname *database_name*

Required. Specifies the database to back up.

--backup-dir *directory*

Optional. Copies all required backup files (metadata files and data files) to the specified directory. You must specify *directory* as an absolute path (not relative). If you do not supply this option, metadata files are created on the Greenplum Database master host in the `$MASTER_DATA_DIRECTORY/backups/YYYYMMDD/YYYYMMDDhhmmss/` directory. Segment hosts create CSV data files in the `<seg_dir>/backups/YYYYMMDD/YYYYMMDDhhmmss/` directory. When you specify a custom backup directory, files are copied to these paths in subdirectories of the backup directory.

You cannot combine this option with the option `--plugin-config`.

--compression-level *level*

Optional. Specifies the gzip compression level (from 1 to 9) used to compress data files. The default is 1. Note that `gpbackup` uses compression by default.

--data-only

Optional. Backs up only the table data into CSV files, but does not backup metadata files needed to recreate the tables and other database objects.

--debug

Optional. Displays verbose debug messages during operation.

--exclude-schema *schema_name*

Optional. Specifies a database schema to exclude from the backup. You can specify this option multiple times to exclude multiple schemas. You cannot combine this option with the option `--include-schema`, `--include-schema-file`, or a table filtering option such as `--include-table`.

See *Filtering the Contents of a Backup or Restore* for more information.

See *Requirements and Limitations* for limitations when leaf partitions of a partitioned table are in different schemas from the root partition.

--exclude-schema-file *file_name*

Optional. Specifies a text file containing a list of schemas to exclude from the backup. Each line in the text file must define a single schema. The file must not include trailing lines. If a schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. You cannot combine this option with the option `--include-schema` or `--include-schema-file`, or a table filtering option such as `--include-table`.

See *Filtering the Contents of a Backup or Restore* for more information.

See *Requirements and Limitations* for limitations when leaf partitions of a partitioned table are in different schemas from the root partition.

--exclude-table *schema.table*

Optional. Specifies a table to exclude from the backup. The table must be in the format `<schema-name>.<table-name>`. If a table or schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. You can specify this option multiple times. You cannot combine this option with the option `--exclude-schema`, `--exclude-schema-file`, or another a table filtering option such as `--include-table`.

You cannot use this option in combination with `--leaf-partition-data`. Although you can specify leaf partition names, `gpbackup` ignores the partition names.

See *Filtering the Contents of a Backup or Restore* for more information.

--exclude-table-file *file_name*

Optional. Specifies a text file containing a list of tables to exclude from the backup. Each line in the text file must define a single table using the format `<schema-name>.<table-`

name>. The file must not include trailing lines. If a table or schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. You cannot combine this option with the option `--exclude-schema`, `--exclude-schema-file`, or another a table filtering option such as `--include-table`.

You cannot use this option in combination with `--leaf-partition-data`. Although you can specify leaf partition names in a file specified with `--exclude-table-file`, `gpbackup` ignores the partition names.

See *Filtering the Contents of a Backup or Restore* for more information.

`--include-schema` *`schema_name`*

Optional. Specifies a database schema to include in the backup. You can specify this option multiple times to include multiple schemas. If you specify this option, any schemas that are not included in subsequent `--include-schema` options are omitted from the backup set. You cannot combine this option with the options `--exclude-schema`, `--exclude-schema-file`, `--exclude-schema-file`, `--include-table`, or `--include-table-file`. See *Filtering the Contents of a Backup or Restore* for more information.

`--include-schema-file` *`file_name`*

Optional. Specifies a text file containing a list of schemas to back up. Each line in the text file must define a single schema. The file must not include trailing lines. If a schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. See *Filtering the Contents of a Backup or Restore* for more information.

`--include-table` *`schema.table`*

Optional. Specifies a table to include in the backup. The table must be in the format `<schema-name>.<table-name>`. If a table or schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in single quotes. See *Schema and Table Names* for information about characters that are supported in schema and table names.

You can specify this option multiple times. You cannot combine this option with a schema filtering option such as `--include-schema`, or another table filtering option such as `--exclude-table-file`.

You can also specify the qualified name of a sequence, a view, or a materialized view.

If you specify this option, the utility does not automatically back up dependent objects. You must also explicitly specify dependent objects that are required. For example if you back up a view or a materialized view, you must also back up the tables that the view or materialized view uses. If you back up a table that uses a sequence, you must also back up the sequence.

You can optionally specify a table leaf partition name in place of the table name, to include only specific leaf partitions in a backup with the `--leaf-partition-data` option. When a leaf partition is backed up, the leaf partition data is backed up along with the metadata for the partitioned table.

See *Filtering the Contents of a Backup or Restore* for more information.

`--include-table-file` *`file_name`*

Optional. Specifies a text file containing a list of tables to include in the backup. Each line in the text file must define a single table using the format `<schema-name>.<table-name>`. The file must not include trailing lines. See *Schema and Table Names* for information about characters that are supported in schema and table names.

Any tables not listed in this file are omitted from the backup set. You cannot combine this option with a schema filtering option such as `--include-schema`, or another table filtering option such as `--exclude-table-file`.

You can also specify the qualified name of a sequence, a view, or a materialized view.

If you specify this option, the utility does not automatically back up dependent objects. You must also explicitly specify dependent objects that are required. For example if you back up a view or a materialized view, you must also specify the tables that the view or the materialized view uses. If you specify a table that uses a sequence, you must also specify the sequence.

You can optionally specify a table leaf partition name in place of the table name, to include only specific leaf partitions in a backup with the `--leaf-partition-data` option. When a leaf partition is backed up, the leaf partition data is backed up along with the metadata for the partitioned table.

See *Filtering the Contents of a Backup or Restore* for more information.

`--incremental`

Specify this option to add an incremental backup to an incremental backup set. A backup set is a full backup and one or more incremental backups. The backups in the set must be created with a consistent set of backup options to ensure that the backup set can be used in a restore operation.

By default, `gpbackup` attempts to find the most recent existing backup with a consistent set of options. If the backup is a full backup, the utility creates a backup set. If the backup is an incremental backup, the utility adds the backup to the existing backup set. The incremental backup is added as the latest backup in the backup set. You can specify `--from-timestamp` to override the default behavior.

`--from-timestamp` ***`backup-timestamp`***

Optional. Specifies the timestamp of a backup. The specified backup must have backup options that are consistent with the incremental backup that is being created. If the specified backup is a full backup, the utility creates a backup set. If the specified backup is an incremental backup, the utility adds the incremental backup to the existing backup set.

You must specify `--leaf-partition-data` with this option. You cannot combine this option with `--data-only` or `--metadata-only`.

A backup is not created and the utility returns an error if the backup cannot add the backup to an existing incremental backup set or cannot use the backup to create a backup set.

For information about creating and using incremental backups, see *Creating and Using Incremental Backups with gpbackup and gprestore*.

`--jobs` ***`int`***

Optional. Specifies the number of jobs to run in parallel when backing up tables. By default, `gpbackup` uses 1 job (database connection). Increasing this number can improve the speed of backing up data. When running multiple jobs, each job backs up tables in a separate transaction. For example, if you specify `--jobs 2`, the utility creates two processes, each process starts a single transaction, and the utility backs up the tables in parallel using the two processes.

> **Important:** If you specify a value higher than 1, the database must be in a quiescent state while the utility acquires a lock on the tables that are being backed up. If database operations are being performed on tables that are being backed up during the table locking process, consistency between tables that are backed up in different transactions cannot be guaranteed.

You cannot use this option in combination with the options `--metadata-only`, `--single-data-file`, or `--plugin-config`.

--leaf-partition-data

Optional. For partitioned tables, creates one data file per leaf partition instead of one data file for the entire table (the default). Using this option also enables you to specify individual leaf partitions to include in a backup, with the `--include-table-file` option. You cannot use this option in combination with `--exclude-table-file` or `--exclude-table`.

--metadata-only

Optional. Creates only the metadata files (DDL) needed to recreate the database objects, but does not back up the actual table data.

--no-compression

Optional. Do not compress the table data CSV files.

--plugin-config *config-file_location*

Specify the location of the `gpbackup` plugin configuration file, a YAML-formatted text file. The file contains configuration information for the plugin application that `gpbackup` uses during the backup operation.

If you specify the `--plugin-config` option when you back up a database, you must specify this option with configuration information for a corresponding plugin application when you restore the database from the backup.

You cannot combine this option with the option `--backup-dir`.

For information about using storage plugin applications, see *Using gpbackup Storage Plugins*.

--quiet

Optional. Suppress all non-warning, non-error log messages.

--single-data-file

Optional. Create a single data file on each segment host for all tables backed up on that segment. By default, each `gpbackup` creates one compressed CSV file for each table that is backed up on the segment.

> **Note:** If you use the `--single-data-file` option to combine table backups into a single file per segment, you cannot set the `gprestore` option `--jobs` to a value higher than 1 to perform a parallel restore operation.

--verbose

Optional. Print verbose log messages.

--version

Optional. Print the version number and exit.

--with-stats

Optional. Include query plan statistics in the backup set.

--help

Displays the online help.

Return Codes

One of these codes is returned after `gpbackup` completes.

- **0** – Backup completed with no problems.
- **1** – Backup completed with non-fatal errors. See log file for more information.
- **2** – Backup failed with a fatal error. See log file for more information.

Schema and Table Names

When specifying the table filtering option `--include-table` or `--include-table-file` to list tables to be backed up, the `gpbackup` utility supports backing up schemas or tables when the name contains upper-case characters or these special characters.

```
~ # $ % ^ & * ( ) _ - + [ ] { } > < \ | ; : / ? ! ,
```

If a name contains an upper-case or special character and is specified on the command line with `--include-table`, the name must be enclosed in single quotes.

```
gpbackup --dbname test --include-table 'my#1schema'.'my_$42_Table'
```

When the table is listed in a file for use with `--include-table-file`, single quotes are not required. For example, this is the contents of a text file that is used with `--include-table-file` to back up two tables.

```
my#1schema.my_$42_Table
my#1schema.my_$590_Table
```

> **Note:** The `--include-table` and `--include-table-file` options do not support schema or table names that contain the character double quote (`"`), period (`.`) , newline (`\n`), or space ( ).

Examples

Backup all schemas and tables in the "demo" database, including global Greenplum Database system objects statistics:

```
$ gpbackup --dbname demo
```

Backup all schemas and tables in the "demo" database except for the "twitter" schema:

```
$ gpbackup --dbname demo --exclude-schema twitter
```

Backup only the "twitter" schema in the "demo" database:

```
$ gpbackup --dbname demo --include-schema twitter
```

Backup all schemas and tables in the "demo" database, including global Greenplum Database system objects and query statistics, and copy all backup files to the `/home/gpadmin/backup` directory:

```
$ gpbackup --dbname demo --with-stats --backup-dir /home/gpadmin/backup
```

This example uses `--include-schema` with `--exclude-table` to back up a schema except for a single table.

```
$ gpbackup --dbname demo --include-schema mydata --exclude-table
 mydata.addresses
```

You cannot use the option `--exclude-schema` with a table filtering option such as `--include-table`.

See Also

`gprestore`, *Parallel Backup with gpbackup and gprestore* and *Using the S3 Storage Plugin with gpbackup and gprestore*

gpcheckcat

The `gpcheckcat` utility tests Greenplum Database catalog tables for inconsistencies.

The utility is in `$GPHOME/bin/lib`.

Synopsis

```
gpcheckcat [ options] [ dbname]

  Options:
     -g dir
     -p port
     -P password
     -U user_name
     -S {none | only}
     -O
     -R test_name
     -C catalog_name
     -B parallel_processes
     -v
     -A

gpcheckcat  -l

gpcheckcat -?
```

Description

The `gpcheckcat` utility runs multiple tests that check for database catalog inconsistencies. Some of the tests cannot be run concurrently with other workload statements or the results will not be usable. Restart the database in restricted mode when running `gpcheckcat`, otherwise `gpcheckcat` might report inconsistencies due to ongoing database operations rather than the actual number of inconsistencies. If you run `gpcheckcat` without stopping database activity, run it with `-O` option.

> **Note:** Any time you run the utility, it checks for and deletes orphaned, temporary database schemas (temporary schemas without a session ID) in the specified databases. The utility displays the results of the orphaned, temporary schema check on the command line and also logs the results.

Catalog inconsistencies are inconsistencies that occur between Greenplum Database system tables. In general, there are three types of inconsistencies:

- Inconsistencies in system tables at the segment level. For example, an inconsistency between a system table that contains table data and a system table that contains column data. As another, a system table that contains duplicates in a column that should to be unique.
- Inconsistencies between same system table across segments. For example, a system table is missing row on one segment, but other segments have this row. As another example, the values of specific row column data are different across segments, such as table owner or table access privileges.
- Inconsistency between a catalog table and the filesystem. For example, a file exists in database directory, but there is no entry for it in the pg_class table.

Options

`-A`

Run `gpcheckcat` on all databases in the Greenplum Database installation.

`-B parallel_processes`

The number of processes to run in parallel.

The `gpcheckcat` utility attempts to determine the number of simultaneous processes (the batch size) to use. The utility assumes it can use a buffer with a minimum of 20MB for each process. The maximum number of parallel processes is the number of Greenplum

Database segment instances. The utility displays the number of parallel processes that it uses when it starts checking the catalog.

> **Note:** The utility might run out of memory if the number of errors returned exceeds the buffer size. If an out of memory error occurs, you can lower the batch size with the `-B` option. For example, if the utility displays a batch size of 936 and runs out of memory, you can specify `-B 468` to run 468 processes in parallel.

`-C` *`catalog_table`*

Run cross consistency, foreign key, and ACL tests for the specified catalog table.

`-g` *`data_directory`*

Generate SQL scripts to fix catalog inconsistencies. The scripts are placed in *data_directory*.

`-l`

List the `gpcheckcat` tests.

`-O`

Run only the `gpcheckcat` tests that can be run in online (not restricted) mode.

`-p` *`port`*

This option specifies the port that is used by the Greenplum Database.

`-P` *`password`*

The password of the user connecting to Greenplum Database.

`-R` *`test_name`*

Specify a test to run. Some tests can be run only when Greenplum Database is in restricted mode.

These are the tests that can be performed:

`acl` - Cross consistency check for access control privileges

`aoseg_table` - Check that the vertical partition information (vpinfo) on segment instances is consistent with `pg_attribute` (checks only append-optimized, column storage tables in the database)

`duplicate` - Check for duplicate entries

`foreign_key` - Check foreign keys

`inconsistent` - Cross consistency check for master segment inconsistency

`missing_extraneous` - Cross consistency check for missing or extraneous entries

`owner` - Check table ownership that is inconsistent with the master database

`orphaned_toast_tables` - Check for orphaned TOAST tables.

> **Note:** There are several ways a TOAST table can become orphaned where a repair script cannot be generated and a manual catalog change is required. One way is if the `reltoastrelid` entry in *pg_class* points to an incorrect TOAST table (a TOAST table mismatch). Another way is if both the `reltoastrelid` in *pg_class* is missing and the `pg_depend` entry is missing (a double orphan TOAST table). If a manual catalog change is needed, `gpcheckcat` will display detailed steps you can follow to update the catalog. Contact Pivotal Support if you need help with the catalog change.

`part_integrity` - Check *pg_partition* branch integrity, partition with OIDs, partition distribution policy

`part_constraint` - Check constraints on partitioned tables

`unique_index_violation` - Check tables that have columns with the unique index constraint for duplicate entries

`dependency` - Check for dependency on non-existent objects (restricted mode only)

`distribution_policy` - Check constraints on randomly distributed tables (restricted mode only)

`namespace` - Check for schemas with a missing schema definition (restricted mode only)

`pgclass` - Check *pg_class* entry that does not have any corresponding *pg_attribute* entry (restricted mode only)

`-S {none | only}`

Specify this option to control the testing of catalog tables that are shared across all databases in the Greenplum Database installation, such as *pg_database*.

The value `none` disables testing of shared catalog tables. The value `only` tests only the shared catalog tables.

`-U user_name`

The user connecting to Greenplum Database.

`-? (help)`

Displays the online help.

`-v (verbose)`

Displays detailed information about the tests that are performed.

Notes

The utility identifies tables with missing attributes and displays them in various locations in the output and in a non-standardized format. The utility also displays a summary list of tables with missing attributes in the format *`database.schema.table.segment_id`* after the output information is displayed.

If `gpcheckcat` detects inconsistent OID (Object ID) information, it generates one or more verification files that contain an SQL query. You can run the SQL query to see details about the OID inconsistencies and investigate the inconsistencies. The files are generated in the directory where `gpcheckcat` is invoked.

This is the format of the file:

```
gpcheckcat.verify.dbname.catalog_table_name.test_name.TIMESTAMP.sql
```

This is an example verification filename created by `gpcheckcat` when it detects inconsistent OID (Object ID) information in the catalog table *pg_type* in the database `mydb`:

```
gpcheckcat.verify.mydb.pg_type.missing_extraneous.20150420102715.sql
```

This is an example query from a verification file:

```
SELECT *
  FROM (
       SELECT relname, oid FROM pg_class WHERE reltype
         IN (1305822,1301043,1301069,1301095)
       UNION ALL
       SELECT relname, oid FROM gp_dist_random('pg_class') WHERE reltype
         IN (1305822,1301043,1301069,1301095)
       ) alltyprelids
  GROUP BY relname, oid ORDER BY count(*) desc ;
```

gpcheckperf

Verifies the baseline hardware performance of the specified hosts.

Synopsis

```
gpcheckperf -d test_directory [-d test_directory ...]
    {-f hostfile_gpcheckperf | - h hostname [-h hostname ...]}
    [-r ds] [-B block_size] [-S file_size] [-D] [-v|-V]

gpcheckperf -d temp_directory
    {-f hostfile_gpchecknet | - h hostname [-h hostname ...]}
    [ -r n|N|M [--duration time] [--netperf] ] [-D] [-v | -V]

gpcheckperf -?

gpcheckperf --version
```

Description

The `gpcheckperf` utility starts a session on the specified hosts and runs the following performance tests:

- **Disk I/O Test (dd test)** — To test the sequential throughput performance of a logical disk or file system, the utility uses the **dd** command, which is a standard UNIX utility. It times how long it takes to write and read a large file to and from disk and calculates your disk I/O performance in megabytes (MB) per second. By default, the file size that is used for the test is calculated at two times the total random access memory (RAM) on the host. This ensures that the test is truly testing disk I/O and not using the memory cache.
- **Memory Bandwidth Test (stream)** — To test memory bandwidth, the utility uses the STREAM benchmark program to measure sustainable memory bandwidth (in MB/s). This tests that your system is not limited in performance by the memory bandwidth of the system in relation to the computational performance of the CPU. In applications where the data set is large (as in Greenplum Database), low memory bandwidth is a major performance issue. If memory bandwidth is significantly lower than the theoretical bandwidth of the CPU, then it can cause the CPU to spend significant amounts of time waiting for data to arrive from system memory.
- **Network Performance Test (gpnetbench*)** — To test network performance (and thereby the performance of the Greenplum Database interconnect), the utility runs a network benchmark program that transfers a 5 second stream of data from the current host to each remote host included in the test. The data is transferred in parallel to each remote host and the minimum, maximum, average and median network transfer rates are reported in megabytes (MB) per second. If the summary transfer rate is slower than expected (less than 100 MB/s), you can run the network test serially using the `-r n` option to obtain per-host results. To run a full-matrix bandwidth test, you can specify `-r M` which will cause every host to send and receive data from every other host specified. This test is best used to validate if the switch fabric can tolerate a full-matrix workload.

To specify the hosts to test, use the `-f` option to specify a file containing a list of host names, or use the `-h` option to name single host names on the command-line. If running the network performance test, all entries in the host file must be for network interfaces within the same subnet. If your segment hosts have multiple network interfaces configured on different subnets, run the network test once for each subnet.

You must also specify at least one test directory (with `-d`). The user who runs `gpcheckperf` must have write access to the specified test directories on all remote hosts. For the disk I/O test, the test directories should correspond to your segment data directories (primary and/or mirrors). For the memory bandwidth and network tests, a temporary directory is required for the test program files.

Before using `gpcheckperf`, you must have a trusted host setup between the hosts involved in the performance test. You can use the utility `gpssh-exkeys` to update the known host files and exchange public keys between hosts if you have not done so already. Note that `gpcheckperf` calls to `gpssh` and `gpscp`, so these Greenplum utilities must also be in your `$PATH`.

Options

`-B block_size`

Specifies the block size (in KB or MB) to use for disk I/O test. The default is 32KB, which is the same as the Greenplum Database page size. The maximum block size is 1 MB.

-d *test_directory*

For the disk I/O test, specifies the file system directory locations to test. You must have write access to the test directory on all hosts involved in the performance test. You can use the `-d` option multiple times to specify multiple test directories (for example, to test disk I/O of your primary and mirror data directories).

-d *temp_directory*

For the network and stream tests, specifies a single directory where the test program files will be copied for the duration of the test. You must have write access to this directory on all hosts involved in the test.

-D (display per-host results)

Reports performance results for each host for the disk I/O tests. The default is to report results for just the hosts with the minimum and maximum performance, as well as the total and average performance of all hosts.

--duration *time*

Specifies the duration of the network test in seconds (s), minutes (m), hours (h), or days (d). The default is 15 seconds.

-f *hostfile_gpcheckperf*

For the disk I/O and stream tests, specifies the name of a file that contains one host name per host that will participate in the performance test. The host name is required, and you can optionally specify an alternate user name and/or SSH port number per host. The syntax of the host file is one host per line as follows:

```
[username@]hostname[:ssh_port]
```

-f *hostfile_gpchecknet*

For the network performance test, all entries in the host file must be for host addresses within the same subnet. If your segment hosts have multiple network interfaces configured on different subnets, run the network test once for each subnet. For example (a host file containing segment host address names for interconnect subnet 1):

```
sdw1-1
sdw2-1
sdw3-1
```

-h *hostname*

Specifies a single host name (or host address) that will participate in the performance test. You can use the `-h` option multiple times to specify multiple host names.

--netperf

Specifies that the `netperf` binary should be used to perform the network test instead of the Greenplum network test. To use this option, you must download `netperf` from *http://www.netperf.org* and install it into `$GPHOME/bin/lib` on all Greenplum hosts (master and segments).

-r ds{n|N|M}

Specifies which performance tests to run. The default is `dsn`:

- Disk I/O test (d)
- Stream test (s)
- Network performance test in sequential (`n`), parallel (`N`), or full-matrix (`M`) mode. The optional `--duration` option specifies how long (in seconds) to run the network test. To use the parallel (`N`) mode, you must run the test on an *even* number of hosts.

If you would rather use `netperf` (*http://www.netperf.org*) instead of the Greenplum network test, you can download it and install it into `$GPHOME/bin/lib` on all Greenplum hosts (master and segments). You would then specify the optional `--netperf` option to use the `netperf` binary instead of the default `gpnetbench*` utilities.

`-S file_size`

Specifies the total file size to be used for the disk I/O test for all directories specified with `-d`. *file_size* should equal two times total RAM on the host. If not specified, the default is calculated at two times the total RAM on the host where `gpcheckperf` is executed. This ensures that the test is truly testing disk I/O and not using the memory cache. You can specify sizing in KB, MB, or GB.

`-v (verbose) | -V (very verbose)`

Verbose mode shows progress and status messages of the performance tests as they are run. Very verbose mode shows all output messages generated by this utility.

`--version`

Displays the version of this utility.

`-? (help)`

Displays the online help.

Examples

Run the disk I/O and memory bandwidth tests on all the hosts in the file *host_file* using the test directory of */data1* and */data2*:

```
$ gpcheckperf -f hostfile_gpcheckperf -d /data1 -d /data2 -r ds
```

Run only the disk I/O test on the hosts named *sdw1* and *sdw2* using the test directory of */data1*. Show individual host results and run in verbose mode:

```
$ gpcheckperf -h sdw1 -h sdw2 -d /data1 -r d -D -v
```

Run the parallel network test using the test directory of */tmp,* where *hostfile_gpcheck_ic** specifies all network interface host address names within the same interconnect subnet:

```
$ gpcheckperf -f hostfile_gpchecknet_ic1 -r N -d /tmp
$ gpcheckperf -f hostfile_gpchecknet_ic2 -r N -d /tmp
```

Run the same test as above, but use `netperf` instead of the Greenplum network test (note that `netperf` must be installed in `$GPHOME/bin/lib` on all Greenplum hosts):

```
$ gpcheckperf -f hostfile_gpchecknet_ic1 -r N --netperf -d /tmp
$ gpcheckperf -f hostfile_gpchecknet_ic2 -r N --netperf -d /tmp
```

See Also

gpssh, *gpscp*

gpconfig

Sets server configuration parameters on all segments within a Greenplum Database system.

Synopsis

```
gpconfig -c param_name -v value [-m master_value | --masteronly]
       | -r param_name [--masteronly]
       | -l
       [--skipvalidation] [--verbose] [--debug]

gpconfig -s param_name [--file | --file-compare] [--verbose] [--debug]

gpconfig --help
```

Description

The `gpconfig` utility allows you to set, unset, or view configuration parameters from the `postgresql.conf` files of all instances (master, segments, and mirrors) in your Greenplum Database system. When setting a parameter, you can also specify a different value for the master if necessary. For example, parameters such as `max_connections` require a different setting on the master than what is used for the segments. If you want to set or unset a global or master only parameter, use the `--masteronly` option.

`gpconfig` can only be used to manage certain parameters. For example, you cannot use it to set parameters such as `port`, which is required to be distinct for every segment instance. Use the `-l` (list) option to see a complete list of configuration parameters supported by `gpconfig`.

When `gpconfig` sets a configuration parameter in a segment `postgresql.conf` file, the new parameter setting always displays at the bottom of the file. When you use `gpconfig` to remove a configuration parameter setting, `gpconfig` comments out the parameter in all segment `postgresql.conf` files, thereby restoring the system default setting. For example, if you use `gpconfig` to remove (comment out) a parameter and later add it back (set a new value), there will be two instances of the parameter; one that is commented out, and one that is enabled and inserted at the bottom of the `postgresql.conf` file.

After setting a parameter, you must restart your Greenplum Database system or reload the `postgresql.conf` files in order for the change to take effect. Whether you require a restart or a reload depends on the parameter.

For more information about the server configuration parameters, see the *Greenplum Database Reference Guide*.

To show the currently set values for a parameter across the system, use the `-s` option.

`gpconfig` uses the following environment variables to connect to the Greenplum Database master instance and obtain system configuration information:

- PGHOST
- PGPORT
- PGUSER
- PGPASSWORD
- PGDATABASE

Options

-c | --change *param_name*

Changes a configuration parameter setting by adding the new setting to the bottom of the `postgresql.conf` files.

-v | --value *value*

The value to use for the configuration parameter you specified with the `-c` option. By default, this value is applied to all segments, their mirrors, the master, and the standby master.

The utility correctly quotes the value when adding the setting to the `postgresql.conf` files.

To set the value to an empty string, enter empty single quotes (`''`).

`-m | --mastervalue` *`master_value`*

The master value to use for the configuration parameter you specified with the `-c` option. If specified, this value only applies to the master and standby master. This option can only be used with `-v`.

`--masteronly`

When specified, `gpconfig` will only edit the master `postgresql.conf` file.

`-r | --remove` *`param_name`*

Removes a configuration parameter setting by commenting out the entry in the `postgresql.conf` files.

`-l | --list`

Lists all configuration parameters supported by the `gpconfig` utility.

`-s | --show` *`param_name`*

Shows the value for a configuration parameter used on all instances (master and segments) in the Greenplum Database system. If there is a difference in a parameter value among the instances, the utility displays an error message. Running `gpconfig` with the `-s` option reads parameter values directly from the database, and not the `postgresql.conf` file. If you are using `gpconfig` to set configuration parameters across all segments, then running `gpconfig -s` to verify the changes, you might still see the previous (old) values. You must reload the configuration files (`gpstop -u`) or restart the system (`gpstop -r`) for changes to take effect.

`--file`

For a configuration parameter, shows the value from the `postgresql.conf` file on all instances (master and segments) in the Greenplum Database system. If there is a difference in a parameter value among the instances, the utility displays a message. Must be specified with the `-s` option.

For example, the configuration parameter `statement_mem` is set to 64MB for a user with the `ALTER ROLE` command, and the value in the `postgresql.conf` file is 128MB. Running the command `gpconfig -s statement_mem --file` displays 128MB. The command `gpconfig -s statement_mem` run by the user displays 64MB.

Not valid with the `--file-compare` option.

`--file-compare`

For a configuration parameter, compares the current Greenplum Database value with the value in the `postgresql.conf` files on hosts (master and segments). The values in the `postgresql.conf files` represent the value when Greenplum Database is restarted.

If the values are not the same, the utility displays the values from all hosts. If all hosts have the same value, the utility displays a summary report.

Not valid with the `--file` option.

`--skipvalidation`

Overrides the system validation checks of `gpconfig` and allows you to operate on any server configuration parameter, including hidden parameters and restricted parameters that cannot be changed by `gpconfig`. When used with the `-l` option (list), it shows the list of restricted parameters.

> **Warning:** Use extreme caution when setting configuration parameters with this option.

`--verbose`

Displays additional log information during `gpconfig` command execution.

`--debug`

Sets logging output to debug level.

`-? | -h | --help`

Displays the online help.

Examples

Set the `max_connections` setting to 100 on all segments and 10 on the master:

```
gpconfig -c max_connections -v 100 -m 10
```

These examples shows the syntax required due to bash shell string processing.

```
gpconfig -c search_path -v '"\$user",public'
gpconfig -c dynamic_library_path -v '\$libdir'
```

The configuration parameters are added to the `postgresql.conf` file.

```
search_path='"$user",public'
dynamic_library_path='$libdir'
```

Comment out all instances of the `default_statistics_target` configuration parameter, and restore the system default:

```
gpconfig -r default_statistics_target
```

List all configuration parameters supported by `gpconfig`:

```
gpconfig -l
```

Show the values of a particular configuration parameter across the system:

```
gpconfig -s max_connections
```

See Also

gpstop

gpcopy

The `gpcopy` utility copies objects from databases in a source Greenplum Database system to databases in a destination Greenplum Database system.

> **Note:** The `gpcopy` utility is available as a separate download for the commercial release of Pivotal Greenplum Database. See the *Pivotal gpcopy Documentation*.

gpdeletesystem

Deletes a Greenplum Database system that was initialized using `gpinitsystem`.

Synopsis

```
gpdeletesystem [-d master_data_directory] [-B parallel_processes]
   [-f] [-l logfile_directory] [-D]
```

```
gpdeletesystem -?

gpdeletesystem -v
```

Description

The `gpdeletesystem` utility performs the following actions:

- Stop all `postgres` processes (the segment instances and master instance).
- Deletes all data directories.

Before running `gpdeletesystem`:

- Move any backup files out of the master and segment data directories.
- Make sure that Greenplum Database is running.
- If you are currently in a segment data directory, change directory to another location. The utility fails with an error when run from within a segment data directory.

This utility will not uninstall the Greenplum Database software.

Options

-d *master_data_directory*

Specifies the master host data directory. If this option is not specified, the setting for the environment variable `MASTER_DATA_DIRECTORY` is used. If this option is specified, it overrides any setting of `MASTER_DATA_DIRECTORY`. If *master_data_directory* cannot be determined, the utility returns an error.

-B *parallel_processes*

The number of segments to delete in parallel. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances it needs to delete.

-f (force)

Force a delete even if backup files are found in the data directories. The default is to not delete Greenplum Database instances if backup files are present.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-D (debug)

Sets logging level to debug.

-? (help)

Displays the online help.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

Examples

Delete a Greenplum Database system:

```
gpdeletesystem -d /gpdata/gp-1
```

Delete a Greenplum Database system even if backup files are present:

```
gpdeletesystem -d /gpdata/gp-1 -f
```

See Also

gpinitsystem

gpexpand

Expands an existing Greenplum Database across new hosts in the system.

Synopsis

```
gpexpand [{-f|--hosts-file} hosts_file]
        | {-i|--input} input_file [-B batch_size]
        | [{-d | --duration} hh:mm:ss | {-e|--end} 'YYYY-MM-DD hh:mm:ss']
          [-a|-analyze]
          [-n  parallel_processes]
        | {-r|--rollback}
        | {-c|--clean}
        [-v|--verbose] [-s|--silent]
        [{-t|--tardir} directory ]
        [-S|--simple-progress ]

gpexpand -? | -h | --help

gpexpand --version
```

Prerequisites

- You are logged in as the Greenplum Database superuser (`gpadmin`).
- The new segment hosts have been installed and configured as per the existing segment hosts. This involves:
  - Configuring the hardware and OS
  - Installing the Greenplum software
  - Creating the `gpadmin` user account
  - Exchanging SSH keys.
- Enough disk space on your segment hosts to temporarily hold a copy of your largest table.
- When redistributing data, Greenplum Database must be running in production mode. Greenplum Database cannot be running in restricted mode or in master mode. The `gpstart` options `-R` or `-m` cannot be specified to start Greenplum Database.

  Note: These utilities cannot be run while `gpexpand` is performing segment initialization.

  - `gpbackup`
  - `gpcheckcat`
  - `gpconfig`
  - `gppkg`
  - `gprestore`

  Important: When expanding a Greenplum Database system, you must disable Greenplum interconnect proxies before adding new hosts and segment instances to the system, and you must update the `gp_interconnect_proxy_addresses` parameter with the newly-added segment instances before you re-enable interconnect proxies. For information about Greenplum interconnect proxies, see *Configuring Proxies for the Greenplum Interconnect*.

For information about preparing a system for expansion, see *Expanding a Greenplum System* in the *Greenplum Database Administrator Guide*.

Description

The `gpexpand` utility performs system expansion in two phases: segment instance initialization and then table data redistribution.

In the initialization phase, gpexpand runs with an input file that specifies data directories, *dbid* values, and other characteristics of the new segment instances. You can create the input file manually, or by following the prompts in an interactive interview.

If you choose to create the input file using the interactive interview, you can optionally specify a file containing a list of expansion system hosts. If your platform or command shell limits the length of the list of hostnames that you can type when prompted in the interview, specifying the hosts with -f may be mandatory.

In addition to initializing the segment instances, the initialization phase performs these actions:

- Creates an expansion schema named *gpexpand* in the postgres database to store the status of the expansion operation, including detailed status for tables.

In the table data redistribution phase, gpexpand redistributes table data to rebalance the data across the old and new segment instances.

> **Note:** Data redistribution should be performed during low-use hours. Redistribution can be divided into batches over an extended period.

To begin the redistribution phase, run gpexpand with either the -d (duration) or -e (end time) options, or with no options. If you specify an end time or duration, then the utility redistributes tables in the expansion schema until the specified end time or duration is reached. If you specify no options, then the utility redistribution phase continues until all tables in the expansion schema are reorganized. Each table is reorganized using ALTER TABLE commands to rebalance the tables across new segments, and to set tables to their original distribution policy. If gpexpand completes the reorganization of all tables, it displays a success message and ends.

> **Note:** This utility uses secure shell (SSH) connections between systems to perform its tasks. In large Greenplum Database deployments, cloud deployments, or deployments with a large number of segments per host, this utility may exceed the host's maximum threshold for unauthenticated connections. Consider updating the SSH MaxStartups and MaxSessions configuration parameters to increase this threshold. For more information about SSH configuration options, refer to the SSH documentation for your Linux distribution.

Options

-a | --analyze

Run ANALYZE to update the table statistics after expansion. The default is to not run ANALYZE.

-B *batch_size*

Batch size of remote commands to send to a given host before making a one-second pause. Default is 16. Valid values are 1-128.

The gpexpand utility issues a number of setup commands that may exceed the host's maximum threshold for unauthenticated connections as defined by MaxStartups in the SSH daemon configuration. The one-second pause allows authentications to be completed before gpexpand issues any more commands.

The default value does not normally need to be changed. However, it may be necessary to reduce the maximum number of commands if gpexpand fails with connection errors such as 'ssh_exchange_identification: Connection closed by remote host.'

-c | --clean

Remove the expansion schema.

-d | --duration *hh:mm:ss*

Duration of the expansion session from beginning to end.

-e | --end '*YYYY-MM-DD hh:mm:ss*'

Ending date and time for the expansion session.

-f | --hosts-file *filename*

Specifies the name of a file that contains a list of new hosts for system expansion. Each line of the file must contain a single host name.

This file can contain hostnames with or without network interfaces specified. The `gpexpand` utility handles either case, adding interface numbers to end of the hostname if the original nodes are configured with multiple network interfaces.

> **Note:** The Greenplum Database segment host naming convention is `sdwN` where `sdw` is a prefix and `N` is an integer. For example, `sdw1`, `sdw2` and so on. For hosts with multiple interfaces, the convention is to append a dash (`-`) and number to the host name. For example, `sdw1-1` and `sdw1-2` are the two interface names for host `sdw1`.

For information about using a hostname or IP address, see *Specifying Hosts using Hostnames or IP Addresses*. Also, see *Using Host Systems with Multiple NICs*.

-i | --input *input_file*

Specifies the name of the expansion configuration file, which contains one line for each segment to be added in the format of:

hostname|*address*|*port*|*datadir*|*dbid*|*content*|*preferred_role*

-n *parallel_processes*

The number of tables to redistribute simultaneously. Valid values are 1 - 96.

Each table redistribution process requires two database connections: one to alter the table, and another to update the table's status in the expansion schema. Before increasing `-n`, check the current value of the server configuration parameter `max_connections` and make sure the maximum connection limit is not exceeded.

-r | --rollback

Roll back a failed expansion setup operation.

-s | --silent

Runs in silent mode. Does not prompt for confirmation to proceed on warnings.

-S | --simple-progress

If specified, the `gpexpand` utility records only the minimum progress information in the Greenplum Database table *gpexpand.expansion_progress*. The utility does not record the relation size information and status information in the table *gpexpand.status_detail*.

Specifying this option can improve performance by reducing the amount of progress information written to the *gpexpand* tables.

[-t | --tardir] *directory*

The fully qualified path to a *directory* on segment hosts where the `gpexpand` utility copies a temporary tar file. The file contains Greenplum Database files that are used to create segment instances. The default directory is the user home directory.

-v | --verbose

Verbose debugging output. With this option, the utility will output all DDL and DML used to expand the database.

--version

Display the utility's version number and exit.

-? | -h | --help

Displays the online help.

Specifying Hosts using Hostnames or IP Addresses

When expanding a Greenplum Database system, you can specify either a hostname or an IP address for the value.

- If you specify a hostname, the resolution of the hostname to an IP address should be done locally for security. For example, you should use entries in a local `/etc/hosts` file to map a hostname to an IP address. The resolution of a hostname to an IP address should not be performed by an external service such as a public DNS server. You must stop the Greenplum system before you change the mapping of a hostname to a different IP address.
- If you specify an IP address, the address should not be changed after the initial configuration. When segment mirroring is enabled, replication from the primary to the mirror segment will fail if the IP address changes from the configured value. For this reason, you should use a hostname when expanding a Greenplum Database system unless you have a specific requirement to use IP addresses.

When expanding a Greenplum system, `gpexpand` populates *`gp_segment_configuration`* catalog table with the new segment instance information. Greenplum Database uses the `address` value of the `gp_segment_configuration` catalog table when looking up host systems for Greenplum interconnect (internal) communication between the master and segment instances and between segment instances, and for other internal communication.

Using Host Systems with Multiple NICs

If host systems are configured with multiple NICs, you can expand a Greenplum Database system to use each NIC as a Greenplum host system. You must ensure that the host systems are configured with sufficient resources to support all the segment instances being added to the host. Also, if you enable segment mirroring, you must ensure that the expanded Greenplum system configuration supports failover if a host system fails. For information about Greenplum Database mirroring schemes, see *Segment Mirroring Configurations*.

For example, this is a `gpexpand` configuration file for a simple Greenplum system. The segment host `gp6s1` and `gp6s2` are configured with two NICs, `-s1` and `-s2`, where the Greenplum Database system uses each NIC as a host system.

```
gp6s1-s2|gp6s1-s2|40001|/data/data1/gpseg2|6|2|p
gp6s2-s1|gp6s2-s1|50000|/data/mirror1/gpseg2|9|2|m
gp6s2-s1|gp6s2-s1|40000|/data/data1/gpseg3|7|3|p
gp6s1-s2|gp6s1-s2|50001|/data/mirror1/gpseg3|8|3|m
```

Examples

Run `gpexpand` with an input file to initialize new segments and create the expansion schema in the postgres database:

```
$ gpexpand -i input_file
```

Run `gpexpand` for sixty hours maximum duration to redistribute tables to new segments:

```
$ gpexpand -d 60:00:00
```

See Also

`gpssh-exkeys`, *Expanding a Greenplum System* in the *Greenplum Database Administrator Guide*

gpfdist

Serves data files to or writes data files out from Greenplum Database segments.

Synopsis

```
gpfdist [-d directory] [-p http_port] [-P last_http_port] [-l log_file]
   [-t timeout] [-S] [-w time] [-v | -V] [-s] [-m max_length]
   [--ssl certificate_path [--sslclean wait_time] ]
   [-c config.yml]

gpfdist -? | --help

gpfdist --version
```

Description

`gpfdist` is Greenplum Database parallel file distribution program. It is used by readable external tables and `gpload` to serve external table files to all Greenplum Database segments in parallel. It is used by writable external tables to accept output streams from Greenplum Database segments in parallel and write them out to a file.

> **Note:** `gpfdist` and `gpload` are compatible only with the Greenplum Database major version in which they are shipped. For example, a `gpfdist` utility that is installed with Greenplum Database 4.x cannot be used with Greenplum Database 5.x or 6.x.

In order for `gpfdist` to be used by an external table, the `LOCATION` clause of the external table definition must specify the external table data using the `gpfdist://` protocol (see the Greenplum Database command `CREATE EXTERNAL TABLE`).

> **Note:** If the `--ssl` option is specified to enable SSL security, create the external table with the `gpfdists://` protocol.

The benefit of using `gpfdist` is that you are guaranteed maximum parallelism while reading from or writing to external tables, thereby offering the best performance as well as easier administration of external tables.

For readable external tables, `gpfdist` parses and serves data files evenly to all the segment instances in the Greenplum Database system when users `SELECT` from the external table. For writable external tables, `gpfdist` accepts parallel output streams from the segments when users `INSERT` into the external table, and writes to an output file.

> **Note:** When `gpfdist` reads data and encounters a data formatting error, the error message includes a row number indicating the location of the formatting error. `gpfdist` attempts to capture the row that contains the error. However, `gpfdist` might not capture the exact row for some formatting errors.

For readable external tables, if load files are compressed using `gzip` or `bzip2` (have a `.gz` or `.bz2` file extension), `gpfdist` uncompresses the data while loading the data (on the fly). For writable external tables, `gpfdist` compresses the data using `gzip` if the target file has a `.gz` extension.

> **Note:** Compression is not supported for readable and writeable external tables when the `gpfdist` utility runs on Windows platforms.

When reading or writing data with the `gpfdist` or `gpfdists` protocol, Greenplum Database includes `X-GP-PROTO` in the HTTP request header to indicate that the request is from Greenplum Database. The utility rejects HTTP requests that do not include `X-GP-PROTO` in the request header.

Most likely, you will want to run `gpfdist` on your ETL machines rather than the hosts where Greenplum Database is installed. To install `gpfdist` on another host, simply copy the utility over to that host and add `gpfdist` to your `$PATH`.

> **Note:** When using IPv6, always enclose the numeric IP address in brackets.

Options

`-d` *`directory`*

The directory from which `gpfdist` will serve files for readable external tables or create output files for writable external tables. If not specified, defaults to the current directory.

`-l` *`log_file`*

The fully qualified path and log file name where standard output messages are to be logged.

`-p` *`http_port`*

The HTTP port on which `gpfdist` will serve files. Defaults to 8080.

`-P` *`last_http_port`*

The last port number in a range of HTTP port numbers (*http_port* to *last_http_port*, inclusive) on which `gpfdist` will attempt to serve files. `gpfdist` serves the files on the first port number in the range to which it successfully binds.

`-t` *`timeout`*

Sets the time allowed for Greenplum Database to establish a connection to a `gpfdist` process. Default is 5 seconds. Allowed values are 2 to 7200 seconds (2 hours). May need to be increased on systems with a lot of network traffic.

`-m` *`max_length`*

Sets the maximum allowed data row length in bytes. Default is 32768. Should be used when user data includes very wide rows (or when `line too long` error message occurs). Should not be used otherwise as it increases resource allocation. Valid range is 32K to 256MB. (The upper limit is 1MB on Windows systems.)

> **Note:** Memory issues might occur if you specify a large maximum row length and run a large number of `gpfdist` concurrent connections. For example, setting this value to the maximum of 256MB with 96 concurrent `gpfdist` processes requires approximately 24GB of memory (`(96 + 1) x 246MB`).

`-s`

Enables simplified logging. When this option is specified, only messages with `WARN` level and higher are written to the `gpfdist` log file. `INFO` level messages are not written to the log file. If this option is not specified, all `gpfdist` messages are written to the log file.

You can specify this option to reduce the information written to the log file.

`-S (use O_SYNC)`

Opens the file for synchronous I/O with the `O_SYNC` flag. Any writes to the resulting file descriptor block `gpfdist` until the data is physically written to the underlying hardware.

`-w` *`time`*

Sets the number of seconds that Greenplum Database delays before closing a target file such as a named pipe. The default value is 0, no delay. The maximum value is 7200 seconds (2 hours).

For a Greenplum Database with multiple segments, there might be a delay between segments when writing data from different segments to the file. You can specify a time to wait before Greenplum Database closes the file to ensure all the data is written to the file.

`--ssl` *`certificate_path`*

Adds SSL encryption to data transferred with `gpfdist`. After executing `gpfdist` with the `--ssl` *`certificate_path`* option, the only way to load data from this file server is with the `gpfdist://` protocol. For information on the `gpfdist://` protocol, see "Loading and Unloading Data" in the *Greenplum Database Administrator Guide*.

The location specified in *certificate_path* must contain the following files:

- The server certificate file, `server.crt`
- The server private key file, `server.key`

- The trusted certificate authorities, `root.crt`

The root directory (`/`) cannot be specified as *certificate_path*.

`--sslclean` *`wait_time`*

When the utility is run with the `--ssl` option, sets the number of seconds that the utility delays before closing an SSL session and cleaning up the SSL resources after it completes writing data to or from a Greenplum Database segment. The default value is 0, no delay. The maximum value is 500 seconds. If the delay is increased, the transfer speed decreases.

In some cases, this error might occur when copying large amounts of data: `gpfdist server closed connection`. To avoid the error, you can add a delay, for example `--sslclean 5`.

`-c` *`config.yaml`*

Specifies rules that `gpfdist` uses to select a transform to apply when loading or extracting data. The `gpfdist` configuration file is a YAML 1.1 document.

For information about the file format, see *Configuration File Format* in the *Greenplum Database Administrator Guide*. For information about configuring data transformation with `gpfdist`, see *Transforming External Data with gpfdist and gpload* in the *Greenplum Database Administrator Guide*.

This option is not available on Windows platforms.

`-v (verbose)`

Verbose mode shows progress and status messages.

`-V (very verbose)`

Verbose mode shows all output messages generated by this utility.

`-? (help)`

Displays the online help.

`--version`

Displays the version of this utility.

Notes

The server configuration parameter *`verify_gpfdists_cert`* controls whether SSL certificate authentication is enabled when Greenplum Database communicates with the `gpfdist` utility to either read data from or write data to an external data source. You can set the parameter value to `false` to disable authentication when testing the communication between the Greenplum Database external table and the `gpfdist` utility that is serving the external data. If the value is `false`, these SSL exceptions are ignored:

- The self-signed SSL certificate that is used by `gpfdist` is not trusted by Greenplum Database.
- The host name contained in the SSL certificate does not match the host name that is running `gpfdist`.

> **Warning:** Disabling SSL certificate authentication exposes a security risk by not validating the `gpfdists` SSL certificate.

If the `gpfdist` utility hangs with no read or write activity occurring, you can generate a core dump the next time a hang occurs to help debug the issue. Set the environment variable `GPFDIST_WATCHDOG_TIMER` to the number of seconds of no activity to wait before `gpfdist` is forced to exit. When the environment variable is set and `gpfdist` hangs, the utility aborts after the specified number of seconds, creates a core dump, and sends abort information to the log file.

This example sets the environment variable on a Linux system so that `gpfdist` exits after 300 seconds (5 minutes) of no activity.

```
export GPFDIST_WATCHDOG_TIMER=300
```

Examples

To serve files from a specified directory using port 8081 (and start `gpfdist` in the background):

```
gpfdist -d /var/load_files -p 8081 &
```

To start `gpfdist` in the background and redirect output and errors to a log file:

```
gpfdist -d /var/load_files -p 8081 -l /home/gpadmin/log &
```

To stop `gpfdist` when it is running in the background:

--First find its process id:

```
ps ax | grep gpfdist
```

--Then kill the process, for example:

```
kill 3456
```

See Also

gpload, `CREATE EXTERNAL TABLE` in the *Greenplum Database Reference Guide*

gpinitstandby

Adds and/or initializes a standby master host for a Greenplum Database system.

Synopsis

```
gpinitstandby { -s standby_hostname [-P port] | -r | -n } [-a] [-q]
    [-D] [-S standby_data_directory] [-l logfile_directory]

gpinitstandby -v

gpinitstandby -?
```

Description

The `gpinitstandby` utility adds a backup, standby master instance to your Greenplum Database system. If your system has an existing standby master instance configured, use the `-r` option to remove it before adding the new standby master instance.

Before running this utility, make sure that the Greenplum Database software is installed on the standby master host and that you have exchanged SSH keys between the hosts. It is recommended that the master port is set to the same port number on the master host and the standby master host.

This utility should be run on the currently active *primary* master host. See the *Greenplum Database Installation Guide* for instructions.

The utility performs the following steps:

- Updates the Greenplum Database system catalog to remove the existing standby master information (if the `-r` option is supplied)
- Updates the Greenplum Database system catalog to add the new standby master instance information
- Edits the `pg_hba.conf` file of the Greenplum Database master to allow access from the newly added standby master
- Sets up the standby master instance on the alternate master host
- Starts the synchronization process

A backup, standby master instance serves as a 'warm standby' in the event of the primary master becoming non-operational. The standby master is kept up to date by transaction log replication processes (the `walsender` and `walreceiver`), which run on the primary master and standby master hosts and keep the data between the primary and standby master instances synchronized. If the primary master fails, the log replication process is shut down, and the standby master can be activated in its place by using the `gpactivatestandby` utility. Upon activation of the standby master, the replicated logs are used to reconstruct the state of the master instance at the time of the last successfully committed transaction.

The activated standby master effectively becomes the Greenplum Database master, accepting client connections on the master port and performing normal master operations such as SQL command processing and resource management.

> **Important:** If the `gpinitstandby` utility previously failed to initialize the standby master, you must delete the files in the standby master data directory before running `gpinitstandby` again. The standby master data directory is not cleaned up after an initialization failure because it contains log files that can help in determining the reason for the failure.
>
> If an initialization failure occurs, a summary report file is generated in the standby host directory `/tmp`. The report file lists the directories on the standby host that require clean up.

Options

`-a (do not prompt)`

Do not prompt the user for confirmation.

`-D (debug)`

Sets logging level to debug.

`-l` *`logfile_directory`*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

`-n (restart standby master)`

Specify this option to start a Greenplum Database standby master that has been configured but has stopped for some reason.

`-P` *`port`*

This option specifies the port that is used by the Greenplum Database standby master. The default is the same port used by the active Greenplum Database master.

If the Greenplum Database standby master is on the same host as the active master, the ports must be different. If the ports are the same for the active and standby master and the host is the same, the utility returns an error.

`-q (no screen output)`

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

`-r (remove standby master)`

Removes the currently configured standby master instance from your Greenplum Database system.

`-s standby_hostname`

The host name of the standby master host.

`-S` *`standby_data_directory`*

The data directory to use for a new standby master. The default is the same directory used by the active master.

If the standby master is on the same host as the active master, a different directory must be specified using this option.

`-v (show utility version)`

Displays the version, status, last updated date, and checksum of this utility.

-? (help)

Displays the online help.

Examples

Add a standby master instance to your Greenplum Database system and start the synchronization process:

```
gpinitstandby -s host09
```

Start an existing standby master instance and synchronize the data with the current primary master instance:

```
gpinitstandby -n
```

Note: Do not specify the -n and -s options in the same command.

Add a standby master instance to your Greenplum Database system specifying a different port:

```
gpinitstandby -s myhost -P 2222
```

If you specify the same host name as the active Greenplum Database master, you must also specify a different port number with the `-P` option and a standby data directory with the `-S` option.

Remove the existing standby master from your Greenplum system configuration:

```
gpinitstandby -r
```

See Also

gpinitsystem, *gpaddmirrors*, *gpactivatestandby*

gpinitsystem

Initializes a Greenplum Database system using configuration parameters specified in the `gpinitsystem_config` file.

Synopsis

```
gpinitsystem -c cluster_configuration_file
             [-h hostfile_gpinitsystem]
             [-B parallel_processes]
             [-p postgresql_conf_param_file]
             [-s standby_master_host
                 [-P standby_master_port]
                 [-S standby_master_datadir | --
standby_datadir=standby_master_datadir]]
             [--ignore-warnings]
             [-m number | --max_connections=number]
             [-b size | --shared_buffers=size]
             [-n locale | --locale=locale] [--lc-collate=locale]
             [--lc-ctype=locale] [--lc-messages=locale]
             [--lc-monetary=locale] [--lc-numeric=locale]
             [--lc-time=locale] [-e password | --su_password=password]
             [--mirror-mode={group|spread}] [-a] [-q] [-l logfile_directory]
 [-D]
             [-I input_configuration_file]
             [-O output_configuration_file]
```

```
gpinitsystem -v | --version

gpinitsystem -? | --help
```

Description

The `gpinitsystem` utility creates a Greenplum Database instance or writes an input configuration file using the values defined in a cluster configuration file and any command-line options that you provide. See *Initialization Configuration File Format* for more information about the configuration file. Before running this utility, make sure that you have installed the Greenplum Database software on all the hosts in the array.

With the *`-O output_configuration_file`* option, `gpinitsystem` writes all provided configuration information to the specified output file. This file can be used with the `-I` option to create a new cluster or re-create a cluster from a backed up configuration. See *Initialization Configuration File Format* for more information.

In a Greenplum Database DBMS, each database instance (the master instance and all segment instances) must be initialized across all of the hosts in the system in such a way that they can all work together as a unified DBMS. The `gpinitsystem` utility takes care of initializing the Greenplum master and each segment instance, and configuring the system as a whole.

Before running `gpinitsystem`, you must set the `$GPHOME` environment variable to point to the location of your Greenplum Database installation on the master host and exchange SSH keys between all host addresses in the array using `gpssh-exkeys`.

This utility performs the following tasks:

- Verifies that the parameters in the configuration file are correct.
- Ensures that a connection can be established to each host address. If a host address cannot be reached, the utility will exit.
- Verifies the locale settings.
- Displays the configuration that will be used and prompts the user for confirmation.
- Initializes the master instance.
- Initializes the standby master instance (if specified).
- Initializes the primary segment instances.
- Initializes the mirror segment instances (if mirroring is configured).
- Configures the Greenplum Database system and checks for errors.
- Starts the Greenplum Database system.

> **Note:** This utility uses secure shell (SSH) connections between systems to perform its tasks. In large Greenplum Database deployments, cloud deployments, or deployments with a large number of segments per host, this utility may exceed the host's maximum threshold for unauthenticated connections. Consider updating the SSH `MaxStartups` and `MaxSessions` configuration parameters to increase this threshold. For more information about SSH configuration options, refer to the SSH documentation for your Linux distribution.

Options

`-a`

Do not prompt the user for confirmation.

`-B` *`parallel_processes`*

The number of segments to create in parallel. If not specified, the utility will start up to 4 parallel processes at a time.

`-c` *`cluster_configuration_file`*

Required. The full path and filename of the configuration file, which contains all of the defined parameters to configure and initialize a new Greenplum Database

system. See *Initialization Configuration File Format* for a description of this file. You must provide either the `-c cluster_configuration_file` option or the `-I input_configuration_file` option to `gpinitsystem`.

`-D`

Sets log output level to debug.

`-h hostfile_gpinitsystem`

Optional. The full path and filename of a file that contains the host addresses of your segment hosts. If not specified on the command line, you can specify the host file using the `MACHINE_LIST_FILE` parameter in the *gpinitsystem_config* file.

`-I input_configuration_file`

The full path and filename of an input configuration file, which defines the Greenplum Database host systems, the master instance and segment instances on the hosts, using the `QD_PRIMARY_ARRAY`, `PRIMARY_ARRAY`, and `MIRROR_ARRAY` parameters. The input configuration file is typically created by using `gpinitsystem` with the `-O output_configuration_file` option. Edit those parameters in order to initialize a new cluster or re-create a cluster from a backed up configuration. You must provide either the `-c cluster_configuration_file` option or the `-I input_configuration_file` option to `gpinitsystem`.

`--ignore-warnings`

Controls the value returned by `gpinitsystem` when warnings or an error occurs. The utility returns 0 if system initialization completes without warnings. If only warnings occur, system initialization completes and the system is operational.

With this option, `gpinitsystem` also returns 0 if warnings occurred during system initialization, and returns a non-zero value if a fatal error occurs.

If this option is not specified, `gpinitsystem` returns 1 if initialization completes with warnings, and returns value of 2 or greater if a fatal error occurs.

See the `gpinitsystem` log file for warning and error messages.

`-n locale | --locale=locale`

Sets the default locale used by Greenplum Database. If not specified, the `LC_ALL`, `LC_COLLATE`, or `LANG` environment variable of the master host determines the locale. If these are not set, the default locale is `C` (`POSIX`). A locale identifier consists of a language identifier and a region identifier, and optionally a character set encoding. For example, `sv_SE` is Swedish as spoken in Sweden, `en_US` is U.S. English, and `fr_CA` is French Canadian. If more than one character set can be useful for a locale, then the specifications look like this: `en_US.UTF-8` (locale specification and character set encoding). On most systems, the command `locale` will show the locale environment settings and `locale -a` will show a list of all available locales.

`--lc-collate=locale`

Similar to `--locale`, but sets the locale used for collation (sorting data). The sort order cannot be changed after Greenplum Database is initialized, so it is important to choose a collation locale that is compatible with the character set encodings that you plan to use for your data. There is a special collation name of `C` or `POSIX` (byte-order sorting as opposed to dictionary-order sorting). The `C` collation can be used with any character encoding.

`--lc-ctype=locale`

Similar to `--locale`, but sets the locale used for character classification (what character sequences are valid and how they are interpreted). This cannot be changed after Greenplum Database is initialized, so it is important to choose a character classification locale that is compatible with the data you plan to store in Greenplum Database.

`--lc-messages=locale`

Similar to `--locale`, but sets the locale used for messages output by Greenplum Database. The current version of Greenplum Database does not support multiple locales for output messages (all messages are in English), so changing this setting will not have any effect.

`--lc-monetary=`*`locale`*

Similar to `--locale`, but sets the locale used for formatting currency amounts.

`--lc-numeric=`*`locale`*

Similar to `--locale`, but sets the locale used for formatting numbers.

`--lc-time=`*`locale`*

Similar to `--locale`, but sets the locale used for formatting dates and times.

`-l` *`logfile_directory`*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

`-m` *`number`* `| --max_connections=number`

Sets the maximum number of client connections allowed to the master. The default is 250.

`-O` *`output_configuration_file`*

Optional, used during new cluster initialization. This option writes the *cluster_configuration_file* information (used with -c) to the specified *output_configuration_file*. This file defines the Greenplum Database members using the `QD_PRIMARY_ARRAY`, `PRIMARY_ARRAY`, and `MIRROR_ARRAY` parameters. Use this file as a template for the `-I` *input_configuration_file* option. See *Examples* for more information.

`-p` *`postgresql_conf_param_file`*

Optional. The name of a file that contains `postgresql.conf` parameter settings that you want to set for Greenplum Database. These settings will be used when the individual master and segment instances are initialized. You can also set parameters after initialization using the `gpconfig` utility.

`-q`

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

`-b` *`size`* `| --shared_buffers=`*`size`*

Sets the amount of memory a Greenplum server instance uses for shared memory buffers. You can specify sizing in kilobytes (kB), megabytes (MB) or gigabytes (GB). The default is 125MB.

`-s` *`standby_master_host`*

Optional. If you wish to configure a backup master instance, specify the host name using this option. The Greenplum Database software must already be installed and configured on this host.

`-P` *`standby_master_port`*

If you configure a standby master instance with `-s`, specify its port number using this option. The default port is the same as the master port. To run the standby and master on the same host, you must use this option to specify a different port for the standby. The Greenplum Database software must already be installed and configured on the standby host.

`-S` *`standby_master_datadir`* `| --standby_dir=`*`standby_master_datadir`*

If you configure a standby master host with `-s`, use this option to specify its data directory. If you configure a standby on the same host as the master instance, the master and standby must have separate data directories.

`-e` *`superuser_password`* `| --su_password=`*`superuser_password`*

Use this option to specify the password to set for the Greenplum Database superuser account (such as `gpadmin`). If this option is not specified, the default password `gparray` is assigned to the superuser account. You can use the `ALTER ROLE` command to change the password at a later time.

Recommended security best practices:

- Do not use the default password option for production environments.
- Change the password immediately after installation.

`--mirror_mode={group|spread}`

Use this option to specify the placement of mirror segment instances on the segment hosts. The default, `group`, groups the mirror segments for all of a host's primary segments on a single alternate host. `spread` spreads mirror segments for the primary segments on a host across different hosts in the Greenplum Database array. Spreading is only allowed if the number of hosts is greater than the number of segment instances per host. See *Overview of Segment Mirroring* for information about Greenplum Database mirroring strategies.

`-v | --version`

Print the `gpinitsystem` version and exit.

`-? | --help`

Show help about `gpinitsystem` command line arguments, and exit.

Initialization Configuration File Format

`gpinitsystem` requires a cluster configuration file with the following parameters defined. An example initialization configuration file can be found in `$GPHOME/docs/cli_help/gpconfigs/gpinitsystem_config`.

To avoid port conflicts between Greenplum Database and other applications, the Greenplum Database port numbers should not be in the range specified by the operating system parameter `net.ipv4.ip_local_port_range`. For example, if `net.ipv4.ip_local_port_range = 10000 65535`, you could set Greenplum Database base port numbers to these values.

```
PORT_BASE = 6000
MIRROR_PORT_BASE = 7000
```

`ARRAY_NAME`

Required. A name for the cluster you are configuring. You can use any name you like. Enclose the name in quotes if the name contains spaces.

`MACHINE_LIST_FILE`

Optional. Can be used in place of the `-h` option. This specifies the file that contains the list of the segment host address names that comprise the Greenplum Database system. The master host is assumed to be the host from which you are running the utility and should not be included in this file. If your segment hosts have multiple network interfaces, then this file would include all addresses for the host. Give the absolute path to the file.

`SEG_PREFIX`

Required. This specifies a prefix that will be used to name the data directories on the master and segment instances. The naming convention for data directories in a Greenplum Database system is *SEG_PREFIXnumber* where *number* starts with 0 for segment instances (the master is always -1). So for example, if you choose the prefix `gpseg`, your master instance data directory would be named `gpseg-1`, and the segment instances would be named `gpseg0`, `gpseg1`, `gpseg2`, `gpseg3`, and so on.

`PORT_BASE`

Required. This specifies the base number by which primary segment port numbers are calculated. The first primary segment port on a host is set as `PORT_BASE`, and then incremented by one for each additional primary segment on that host. Valid values range from 1 through 65535.

`DATA_DIRECTORY`

Required. This specifies the data storage location(s) where the utility will create the primary segment data directories. The number of locations in the list dictate the number of primary segments that will get created per physical host (if multiple addresses for a host are listed in the host file, the number of segments will be spread evenly across the specified interface addresses). It is OK to list the same data storage area multiple times if you want your data directories created in the same location. The user who runs `gpinitsystem` (for example, the `gpadmin` user) must have permission to write to these directories. For example, this will create six primary segments per host:

```
declare -a DATA_DIRECTORY=(/data1/primary /data1/primary
/data1/primary /data2/primary /data2/primary /data2/primary)
```

`MASTER_HOSTNAME`

Required. The host name of the master instance. This host name must exactly match the configured host name of the machine (run the `hostname` command to determine the correct hostname).

`MASTER_DIRECTORY`

Required. This specifies the location where the data directory will be created on the master host. You must make sure that the user who runs `gpinitsystem` (for example, the `gpadmin` user) has permissions to write to this directory.

`MASTER_PORT`

Required. The port number for the master instance. This is the port number that users and client connections will use when accessing the Greenplum Database system.

`TRUSTED_SHELL`

Required. The shell the `gpinitsystem` utility uses to execute commands on remote hosts. Allowed values are `ssh`. You must set up your trusted host environment before running the `gpinitsystem` utility (you can use `gpssh-exkeys` to do this).

`CHECK_POINT_SEGMENTS`

Required. Maximum distance between automatic write ahead log (WAL) checkpoints, in log file segments (each segment is normally 16 megabytes). This will set the `checkpoint_segments` parameter in the `postgresql.conf` file for each segment instance in the Greenplum Database system.

`ENCODING`

Required. The character set encoding to use. This character set must be compatible with the `--locale` settings used, especially `--lc-collate` and `--lc-ctype`. Greenplum Database supports the same character sets as PostgreSQL.

`DATABASE_NAME`

Optional. The name of a Greenplum Database database to create after the system is initialized. You can always create a database later using the `CREATE DATABASE` command or the `createdb` utility.

`MIRROR_PORT_BASE`

Optional. This specifies the base number by which mirror segment port numbers are calculated. The first mirror segment port on a host is set as `MIRROR_PORT_BASE`, and then incremented by one for each additional mirror segment on that host. Valid values range from 1 through 65535 and cannot conflict with the ports calculated by `PORT_BASE`.

MIRROR_DATA_DIRECTORY

Optional. This specifies the data storage location(s) where the utility will create the mirror segment data directories. There must be the same number of data directories declared for mirror segment instances as for primary segment instances (see the `DATA_DIRECTORY` parameter). The user who runs `gpinitsystem` (for example, the `gpadmin` user) must have permission to write to these directories. For example:

```
declare -a MIRROR_DATA_DIRECTORY=(/data1/mirror
/data1/mirror /data1/mirror /data2/mirror /data2/mirror
/data2/mirror)
```

QD_PRIMARY_ARRAY, PRIMARY_ARRAY, MIRROR_ARRAY

Required when using `input_configuration file` with `-I` option. These parameters specify the Greenplum Database master host, the primary segment, and the mirror segment hosts respectively. During new cluster initialization, use the `gpinitsystem -O` *`output_configuration_file`* to populate `QD_PRIMARY_ARRAY`, `PRIMARY_ARRAY`, `MIRROR_ARRAY`.

To initialize a new cluster or re-create a cluster from a backed up configuration, edit these values in the input configuration file used with the `gpinitsystem -I` *`input_configuration_file`* option. Use one of the following formats to specify the host information:

```
hostname~address~port~data_directory/seg_prefix<segment_id>~dbid~content_id
```

or

```
host~port~data_directory/seg_prefix<segment_id>~dbid~content_id
```

The first format populates the `hostname` and `address` fields in the `gp_segment_configuration` catalog table with the *hostname* and *address* values provided in the input configuration file. The second format populates `hostname` and `address` fields with the same value, derived from *host*.

The Greenplum Database master always uses the value -1 for the segment ID and content ID. For example, *seg_prefix<segment_id>* and *dbid* values for `QD_PRIMARY_ARRAY` use `-1` to indicate the master instance:

```
QD_PRIMARY_ARRAY=mdw~mdw~5432~/gpdata/master/gpseg-1~1~-1
declare -a PRIMARY_ARRAY=(
sdw1~sdw1~40000~/gpdata/data1/gpseg0~2~0
sdw1~sdw1~40001~/gpdata/data2/gpseg1~3~1
sdw2~sdw2~40000~/gpdata/data1/gpseg2~4~2
sdw2~sdw2~40001~/gpdata/data2/gpseg3~5~3
)
declare -a MIRROR_ARRAY=(
sdw2~sdw2~50000~/gpdata/mirror1/gpseg0~6~0
sdw2~sdw2~50001~/gpdata/mirror2/gpseg1~7~1
sdw1~sdw1~50000~/gpdata/mirror1/gpseg2~8~2
sdw1~sdw1~50001~/gpdata/mirror2/gpseg3~9~3
)
```

To re-create a cluster using a known Greenplum Database system configuration, you can edit the segment and content IDs to match the values of the system.

HEAP_CHECKSUM

Optional. This parameter specifies if checksums are enabled for heap data. When enabled, checksums are calculated for heap storage in all databases, enabling Greenplum

Database to detect corruption in the I/O system. This option is set when the system is initialized and cannot be changed later.

The `HEAP_CHECKSUM` option is on by default and turning it off is strongly discouraged. If you set this option to off, data corruption in storage can go undetected and make recovery much more difficult.

To determine if heap checksums are enabled in a Greenplum Database system, you can query the `data_checksums` server configuration parameter with the `gpconfig` management utility:

```
$ gpconfig -s data_checksums
```

`HBA_HOSTNAMES`

Optional. This parameter controls whether `gpinitsystem` uses IP addresses or host names in the `pg_hba.conf` file when updating the file with addresses that can connect to Greenplum Database. The default value is `0`, the utility uses IP addresses when updating the file. When initializing a Greenplum Database system, specify `HBA_HOSTNAMES=1` to have the utility use host names in the `pg_hba.conf` file.

For information about how Greenplum Database resolves host names in the `pg_hba.conf` file, see *Configuring Client Authentication*.

Specifying Hosts using Hostnames or IP Addresses

When initializing a Greenplum Database system with `gpinitsystem`, you can specify segment hosts using either hostnames or IP addresses. For example, you can use hostnames or IP addresses in the file specified with the `-h` option.

- If you specify a hostname, the resolution of the hostname to an IP address should be done locally for security. For example, you should use entries in a local `/etc/hosts` file to map a hostname to an IP address. The resolution of a hostname to an IP address should not be performed by an external service such as a public DNS server. You must stop the Greenplum system before you change the mapping of a hostname to a different IP address.
- If you specify an IP address, the address should not be changed after the initial configuration. When segment mirroring is enabled, replication from the primary to the mirror segment will fail if the IP address changes from the configured value. For this reason, you should use a hostname when initializing a Greenplum Database system unless you have a specific requirement to use IP addresses.

When initializing the Greenplum Database system, `gpinitsystem` uses the initialization information to populate the *`gp_segment_configuration`* catalog table and adds hosts to the `pg_hba.conf` file. By default, the host IP address is added to the file. Specify the `gpinitsystem` configuration file parameter *`HBA_HOSTNAMES`*`=1` to add hostnames to the file.

Greenplum Database uses the `address` value of the `gp_segment_configuration` catalog table when looking up host systems for Greenplum interconnect (internal) communication between the master and segment instances and between segment instances, and for other internal communication.

Examples

Initialize a Greenplum Database system by supplying a cluster configuration file and a segment host address file, and set up a spread mirroring (`--mirror-mode=spread`) configuration:

```
$ gpinitsystem -c gpinitsystem_config -h hostfile_gpinitsystem --mirror-
mode=spread
```

Initialize a Greenplum Database system and set the superuser remote password:

```
$ gpinitsystem -c gpinitsystem_config -h hostfile_gpinitsystem --su-
password=mypassword
```

Initialize a Greenplum Database system with an optional standby master host:

```
$ gpinitsystem -c gpinitsystem_config -h hostfile_gpinitsystem -s host09
```

Initialize a Greenplum Database system and write the provided configuration to an output file, for example `cluster_init.config`:

```
$ gpinitsystem -c gpinitsystem_config -h hostfile_gpinitsystem -O
 cluster_init.config
```

The output file uses the `QD_PRIMARY_ARRAY` and `PRIMARY_ARRAY` parameters to define master and segment hosts:

```
ARRAY_NAME="Greenplum Data Platform"
TRUSTED_SHELL=ssh
CHECK_POINT_SEGMENTS=8
ENCODING=UNICODE
SEG_PREFIX=gpseg
HEAP_CHECKSUM=on
HBA_HOSTNAMES=0
QD_PRIMARY_ARRAY=mdw~mwd.local~5433~/data/master1/gpseg-1~1~-1
declare -a PRIMARY_ARRAY=(
mwd~mwd.local~6001~/data/primary1/gpseg0~2~0
)
declare -a MIRROR_ARRAY=(
mwd~mwd.local~7001~/data/mirror1/gpseg0~3~0
)
```

Initialize a Greenplum Database using an input configuration file (a file that defines the Greenplum Database cluster) using `QD_PRIMARY_ARRAY` and `PRIMARY_ARRAY` parameters:

```
$ gpinitsystem -I cluster_init.config
```

The following example uses a host system configured with mulitple NICs. If host systems are configured with multiple NICs, you can initialize a Greenplum Database system to use each NIC as a Greenplum host system. You must ensure that the host systems are configured with sufficient resources to support all the segment instances being added to the host. Also, if high availability is enabled, you must ensure that the Greenplum system configuration supports failover if a host system fails. For information about Greenplum Database mirroring schemes, see *Segment Mirroring Configurations*.

For this simple master and segment instance configuration, the host system `gp6m` is configured with two NICs `gp6m-1` and `gp6m-2`. In the configuration, the *`QD_PRIMARY_ARRAY`* parameter defines the master segment using `gp6m-1`. The *`PRIMARY_ARRAY`* and *`MIRROR_ARRAY`* parameters use `gp6m-2` to define a primary and mirror segment instance.

```
QD_PRIMARY_ARRAY=gp6m~gp6m-1~5432~/data/master/gpseg-1~1~-1
declare -a PRIMARY_ARRAY=(
gp6m~gp6m-2~40000~/data/data1/gpseg0~2~0
gp6s~gp6s~40000~/data/data1/gpseg1~3~1
)
declare -a MIRROR_ARRAY=(
gp6s~gp6s~50000~/data/mirror1/gpseg0~4~0
gp6m~gp6m-2~50000~/data/mirror1/gpseg1~5~1
)
```

See Also

`gpssh-exkeys`, *`gpdeletesystem`*, *Initializing Greenplum Database*.

gpload

Runs a load job as defined in a YAML formatted control file.

Synopsis

```
gpload -f control_file [-l log_file] [-h hostname] [-p port]
   [-U username] [-d database] [-W] [--gpfdist_timeout seconds]
   [--no_auto_trans] [--max_retries retry_times] [[-v | -V] [-q]] [-D]

gpload -?

gpload --version
```

Requirements

The client machine where `gpload` is executed must have the following:

- The *`gpfdist`* parallel file distribution program installed and in your `$PATH`. This program is located in `$GPHOME/bin` of your Greenplum Database server installation.
- Network access to and from all hosts in your Greenplum Database array (master and segments).
- Network access to and from the hosts where the data to be loaded resides (ETL servers).

Description

`gpload` is a data loading utility that acts as an interface to the Greenplum Database external table parallel loading feature. Using a load specification defined in a YAML formatted control file, `gpload` executes a load by invoking the Greenplum Database parallel file server (*`gpfdist`*), creating an external table definition based on the source data defined, and executing an `INSERT`, `UPDATE` or `MERGE` operation to load the source data into the target table in the database.

> **Note:** `gpfdist` is compatible only with the Greenplum Database major version in which it is shipped. For example, a `gpfdist` utility that is installed with Greenplum Database 4.x cannot be used with Greenplum Database 5.x or 6.x.
>
> **Note:** The Greenplum Database 5.22 and later `gpload` for Linux is compatible with Greenplum Database 6.x. The Greenplum Database 6.x `gpload` for both Linux and Windows is compatible with Greenplum 5.x.
>
> **Note:** `MERGE` and `UPDATE` operations are not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

The operation, including any SQL commands specified in the `SQL` collection of the YAML control file (see *Control File Format*), are performed as a single transaction to prevent inconsistent data when performing multiple, simultaneous load operations on a target table.

Options

`-f` *`control_file`*

> **Required.** A YAML file that contains the load specification details. See *Control File Format*.

`--gpfdist_timeout` *`seconds`*

> Sets the timeout for the `gpfdist` parallel file distribution program to send a response. Enter a value from `0` to `30` seconds (entering "`0`" to disables timeouts). Note that you might need to increase this value when operating on high-traffic networks.

`-l` *`log_file`*

Specifies where to write the log file. Defaults to `~/gpAdminLogs/gpload_YYYYMMDD`. For more information about the log file, see *Log File Format*.

`--no_auto_trans`

Specify `--no_auto_trans` to disable processing the load operation as a single transaction if you are performing a single load operation on the target table.

By default, `gpload` processes each load operation as a single transaction to prevent inconsistent data when performing multiple, simultaneous operations on a target table.

`-q (no screen output)`

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

`-D (debug mode)`

Check for error conditions, but do not execute the load.

`-v (verbose mode)`

Show verbose output of the load steps as they are executed.

`-V (very verbose mode)`

Shows very verbose output.

`-? (show help)`

Show help, then exit.

`--version`

Show the version of this utility, then exit.

Connection Options

`-d *database*`

The database to load into. If not specified, reads from the load control file, the environment variable `$PGDATABASE` or defaults to the current system user name.

`-h *hostname*`

Specifies the host name of the machine on which the Greenplum Database master database server is running. If not specified, reads from the load control file, the environment variable `$PGHOST` or defaults to `localhost`.

`-p *port*`

Specifies the TCP port on which the Greenplum Database master database server is listening for connections. If not specified, reads from the load control file, the environment variable `$PGPORT` or defaults to 5432.

`--max_retries *retry_times*`

Specifies the maximum number of times `gpload` attempts to connect to Greenplum Database after a connection timeout. The default value is `0`, do not attempt to connect after a connection timeout. A negative integer, such as `-1`, specifies an unlimited number of attempts.

`-U *username*`

The database role name to connect as. If not specified, reads from the load control file, the environment variable `$PGUSER` or defaults to the current system user name.

`-W (force password prompt)`

Force a password prompt. If not specified, reads the password from the environment variable `$PGPASSWORD` or from a password file specified by `$PGPASSFILE` or in `~/.pgpass`. If these are not set, then `gpload` will prompt for a password even if `-W` is not supplied.

Control File Format

The `gpload` control file uses the *YAML 1.1* document format and then implements its own schema for defining the various steps of a Greenplum Database load operation. The control file must be a valid YAML document.

The `gpload` program processes the control file document in order and uses indentation (spaces) to determine the document hierarchy and the relationships of the sections to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.

The basic structure of a load control file is:

```
---
VERSION: 1.0.0.1
DATABASE: db_name
USER: db_username
HOST: master_hostname
PORT: master_port
GPLOAD:
   INPUT:
    - SOURCE:
         LOCAL_HOSTNAME:
           - hostname_or_ip
         PORT: http_port
       | PORT_RANGE: [start_port_range, end_port_range]
         FILE:
           - /path/to/input_file
         SSL: true | false
         CERTIFICATES_PATH: /path/to/certificates
    - FULLY_QUALIFIED_DOMAIN_NAME: true | false
    - COLUMNS:
           - field_name: data_type
    - TRANSFORM: 'transformation'
    - TRANSFORM_CONFIG: 'configuration-file-path'
    - MAX_LINE_LENGTH: integer
    - FORMAT: text | csv
    - DELIMITER: 'delimiter_character'
    - ESCAPE: 'escape_character' | 'OFF'
    - NULL_AS: 'null_string'
    - FILL_MISSING_FIELDS: true | false
    - FORCE_NOT_NULL: true | false
    - QUOTE: 'csv_quote_character'
    - HEADER: true | false
    - ENCODING: database_encoding
    - ERROR_LIMIT: integer
    - LOG_ERRORS: true | false
   EXTERNAL:
      - SCHEMA: schema | '%'
   OUTPUT:
    - TABLE: schema.table_name
    - MODE: insert | update | merge
    - MATCH_COLUMNS:
           - target_column_name
    - UPDATE_COLUMNS:
           - target_column_name
    - UPDATE_CONDITION: 'boolean_condition'
    - MAPPING:
              target_column_name: source_column_name | 'expression'
   PRELOAD:
    - TRUNCATE: true | false
    - REUSE_TABLES: true | false
    - STAGING_TABLE: external_table_name
    - FAST_MATCH: true | false
```

```
   SQL:
    - BEFORE: "sql_command"
    - AFTER: "sql_command"
```

VERSION

Optional. The version of the `gpload` control file schema. The current version is 1.0.0.1.

DATABASE

Optional. Specifies which database in the Greenplum Database system to connect to. If not specified, defaults to `$PGDATABASE` if set or the current system user name. You can also specify the database on the command line using the `-d` option.

USER

Optional. Specifies which database role to use to connect. If not specified, defaults to the current user or `$PGUSER` if set. You can also specify the database role on the command line using the `-U` option.

If the user running `gpload` is not a Greenplum Database superuser, then the appropriate rights must be granted to the user for the load to be processed. See the *Greenplum Database Reference Guide* for more information.

HOST

Optional. Specifies Greenplum Database master host name. If not specified, defaults to localhost or `$PGHOST` if set. You can also specify the master host name on the command line using the `-h` option.

PORT

Optional. Specifies Greenplum Database master port. If not specified, defaults to 5432 or `$PGPORT` if set. You can also specify the master port on the command line using the `-p` option.

GPLOAD

Required. Begins the load specification section. A `GPLOAD` specification must have an `INPUT` and an `OUTPUT` section defined.

INPUT

Required. Defines the location and the format of the input data to be loaded. `gpload` will start one or more instances of the *`gpfdist`* file distribution program on the current host and create the required external table definition(s) in Greenplum Database that point to the source data. Note that the host from which you run `gpload` must be accessible over the network by all Greenplum Database hosts (master and segments).

SOURCE

Required. The `SOURCE` block of an `INPUT` specification defines the location of a source file. An `INPUT` section can have more than one `SOURCE` block defined. Each `SOURCE` block defined corresponds to one instance of the *`gpfdist`* file distribution program that will be started on the local machine. Each `SOURCE` block defined must have a `FILE` specification.

For more information about using the `gpfdist` parallel file server and single and multiple `gpfdist` instances, see *Loading and Unloading Data*.

LOCAL_HOSTNAME

Optional. Specifies the host name or IP address of the local machine on which `gpload` is running. If this machine is configured with multiple network interface cards (NICs), you can specify the host name or IP of each individual NIC to allow network traffic to use all NICs simultaneously. The default is to use the local machine's primary host name or IP only.

PORT

Optional. Specifies the specific port number that the *`gpfdist`* file distribution program should use. You can also supply a `PORT_RANGE` to select an available port from the specified range. If both `PORT` and `PORT_RANGE` are defined, then `PORT` takes precedence.

If neither PORT or PORT_RANGE are defined, the default is to select an available port between 8000 and 9000.

If multiple host names are declared in LOCAL_HOSTNAME, this port number is used for all hosts. This configuration is desired if you want to use all NICs to load the same file or set of files in a given directory location.

PORT_RANGE

Optional. Can be used instead of PORT to supply a range of port numbers from which gpload can choose an available port for this instance of the *gpfdist* file distribution program.

FILE

Required. Specifies the location of a file, named pipe, or directory location on the local file system that contains data to be loaded. You can declare more than one file so long as the data is of the same format in all files specified.

If the files are compressed using gzip or bzip2 (have a .gz or .bz2 file extension), the files will be uncompressed automatically (provided that gunzip or bunzip2 is in your path).

When specifying which source files to load, you can use the wildcard character (*) or other C-style pattern matching to denote multiple files. The files specified are assumed to be relative to the current directory from which gpload is executed (or you can declare an absolute path).

SSL

Optional. Specifies usage of SSL encryption. If SSL is set to true, gpload starts the gpfdist server with the --ssl option and uses the gpfdists:// protocol.

CERTIFICATES_PATH

Required when SSL is true; cannot be specified when SSL is false or unspecified. The location specified in CERTIFICATES_PATH must contain the following files:

- The server certificate file, server.crt
- The server private key file, server.key
- The trusted certificate authorities, root.crt

The root directory (/) cannot be specified as CERTIFICATES_PATH.

FULLY_QUALIFIED_DOMAIN_NAME

Optional. Specifies whether gpload resolve hostnames to the fully qualified domain name (FQDN) or the local hostname. If the value is set to true, names are resolved to the FQDN. If the value is set to false, resolution is to the local hostname. The default is false.

A fully qualified domain name might be required in some situations. For example, if the Greenplum Database system is in a different domain than an ETL application that is being accessed by gpload.

COLUMNS

Optional. Specifies the schema of the source data file(s) in the format of *field_name*:*data_type*. The DELIMITER character in the source file is what separates two data value fields (columns). A row is determined by a line feed character (0x0a).

If the input COLUMNS are not specified, then the schema of the output TABLE is implied, meaning that the source data must have the same column order, number of columns, and data format as the target table.

The default source-to-target mapping is based on a match of column names as defined in this section and the column names in the target `TABLE`. This default mapping can be overridden using the `MAPPING` section.

`TRANSFORM`

Optional. Specifies the name of the input transformation passed to `gpload`. For information about XML transformations, see "Loading and Unloading Data" in the *Greenplum Database Administrator Guide*.

`TRANSFORM_CONFIG`

Required when `TRANSFORM` is specified. Specifies the location of the transformation configuration file that is specified in the `TRANSFORM` parameter, above.

`MAX_LINE_LENGTH`

Optional. An integer that specifies the maximum length of a line in the XML transformation data passed to `gpload`.

`FORMAT`

Optional. Specifies the format of the source data file(s) - either plain text (`TEXT`) or comma separated values (`CSV`) format. Defaults to `TEXT` if not specified. For more information about the format of the source data, see *Loading and Unloading Data*.

`DELIMITER`

Optional. Specifies a single ASCII character that separates columns within each row (line) of data. The default is a tab character in TEXT mode, a comma in CSV mode. You can also specify a non- printable ASCII character or a non-printable unicode character, for example: `"\x1B"` or `"\u001B"`. The escape string syntax, `E'`*`character-code`*`'`, is also supported for non-printable characters. The ASCII or unicode character must be enclosed in single quotes. For example: `E'\x1B'` or `E'\u001B'`.

`ESCAPE`

Specifies the single character that is used for C escape sequences (such as `\n`, `\t`, `\100`, and so on) and for escaping data characters that might otherwise be taken as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. The default escape character is a \ (backslash) for text-formatted files and a `"` (double quote) for csv-formatted files, however it is possible to specify another character to represent an escape. It is also possible to disable escaping in text-formatted files by specifying the value `'OFF'` as the escape value. This is very useful for data such as text-formatted web log data that has many embedded backslashes that are not intended to be escapes.

`NULL_AS`

Optional. Specifies the string that represents a null value. The default is `\N` (backslash-N) in `TEXT` mode, and an empty value with no quotations in `CSV` mode. You might prefer an empty string even in `TEXT` mode for cases where you do not want to distinguish nulls from empty strings. Any source data item that matches this string will be considered a null value.

`FILL_MISSING_FIELDS`

Optional. The default value is `false`. When reading a row of data that has missing trailing field values (the row of data has missing data fields at the end of a line or row), Greenplum Database returns an error.

If the value is `true`, when reading a row of data that has missing trailing field values, the values are set to `NULL`. Blank rows, fields with a `NOT NULL` constraint, and trailing delimiters on a line will still report an error.

See the `FILL MISSING FIELDS` clause of the *`CREATE EXTERNAL TABLE`* command.

`FORCE_NOT_NULL`

Optional. In CSV mode, processes each specified column as though it were quoted and hence not a NULL value. For the default null string in CSV mode (nothing between two delimiters), this causes missing values to be evaluated as zero-length strings.

QUOTE

Required when FORMAT is CSV. Specifies the quotation character for CSV mode. The default is double-quote (").

HEADER

Optional. Specifies that the first line in the data file(s) is a header row (contains the names of the columns) and should not be included as data to be loaded. If using multiple data source files, all files must have a header row. The default is to assume that the input files do not have a header row.

ENCODING

Optional. Character set encoding of the source data. Specify a string constant (such as 'SQL_ASCII'), an integer encoding number, or 'DEFAULT' to use the default client encoding. If not specified, the default client encoding is used. For information about supported character sets, see the *Greenplum Database Reference Guide*.

> **Note:** If you *change* the ENCODING value in an existing gpload control file, you must manually drop any external tables that were creating using the previous ENCODING configuration. gpload does not drop and recreate external tables to use the new ENCODING if REUSE_TABLES is set to true.

ERROR_LIMIT

Optional. Enables single row error isolation mode for this load operation. When enabled, input rows that have format errors will be discarded provided that the error limit count is not reached on any Greenplum Database segment instance during input processing. If the error limit is not reached, all good rows will be loaded and any error rows will either be discarded or captured as part of error log information. The default is to abort the load operation on the first error encountered. Note that single row error isolation only applies to data rows with format errors; for example, extra or missing attributes, attributes of a wrong data type, or invalid client encoding sequences. Constraint errors, such as primary key violations, will still cause the load operation to abort if encountered. For information about handling load errors, see *Loading and Unloading Data*.

LOG_ERRORS

Optional when ERROR_LIMIT is declared. Value is either true or false. The default value is false. If the value is true, rows with formatting errors are logged internally when running in single row error isolation mode. You can examine formatting errors with the Greenplum Database built-in SQL function gp_read_error_log('*table_name*'). If formatting errors are detected when loading data, gpload generates a warning message with the name of the table that contains the error information similar to this message.

```
timestamp|WARN|1 bad row, please use GPDB built-in function
 gp_read_error_log('table-name')
   to access the detailed error row
```

If LOG_ERRORS: true is specified, REUSE_TABLES: true must be specified to retain the formatting errors in Greenplum Database error logs. If REUSE_TABLES: true is not specified, the error information is deleted after the gpload operation. Only summary information about formatting errors is returned. You can delete the formatting errors from the error logs with the Greenplum Database function gp_truncate_error_log().

> **Note:** When gpfdist reads data and encounters a data formatting error, the error message includes a row number indicating the location of the formatting error. gpfdist attempts to capture the row that contains the error. However, gpfdist might not capture the exact row for some formatting errors.

For more information about handling load errors, see "Loading and Unloading Data" in the *Greenplum Database Administrator Guide*. For information about the `gp_read_error_log()` function, see the *CREATE EXTERNAL TABLE* command.

EXTERNAL

Optional. Defines the schema of the external table database objects created by `gpload`.

The default is to use the Greenplum Database `search_path`.

SCHEMA

Required when `EXTERNAL` is declared. The name of the schema of the external table. If the schema does not exist, an error is returned.

If `%` (percent character) is specified, the schema of the table name specified by `TABLE` in the `OUTPUT` section is used. If the table name does not specify a schema, the default schema is used.

OUTPUT

Required. Defines the target table and final data column values that are to be loaded into the database.

TABLE

Required. The name of the target table to load into.

MODE

Optional. Defaults to `INSERT` if not specified. There are three available load modes:

INSERT - Loads data into the target table using the following method:

```
INSERT INTO target_table SELECT * FROM input_data;
```

UPDATE - Updates the `UPDATE_COLUMNS` of the target table where the rows have `MATCH_COLUMNS` attribute values equal to those of the input data, and the optional `UPDATE_CONDITION` is true. `UPDATE` is not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

MERGE - Inserts new rows and updates the `UPDATE_COLUMNS` of existing rows where `FOOBAR` attribute values are equal to those of the input data, and the optional `MATCH_COLUMNS` is true. New rows are identified when the `MATCH_COLUMNS` value in the source data does not have a corresponding value in the existing data of the target table. In those cases, the **entire row** from the source file is inserted, not only the `MATCH` and `UPDATE` columns. If there are multiple new `MATCH_COLUMNS` values that are the same, only one new row for that value will be inserted. Use `UPDATE_CONDITION` to filter out the rows to discard. `MERGE` is not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

MATCH_COLUMNS

Required if `MODE` is `UPDATE` or `MERGE`. Specifies the column(s) to use as the join condition for the update. The attribute value in the specified target column(s) must be equal to that of the corresponding source data column(s) in order for the row to be updated in the target table.

UPDATE_COLUMNS

Required if `MODE` is `UPDATE` or `MERGE`. Specifies the column(s) to update for the rows that meet the `MATCH_COLUMNS` criteria and the optional `UPDATE_CONDITION`.

UPDATE_CONDITION

Optional. Specifies a Boolean condition (similar to what you would declare in a `WHERE` clause) that must be met in order for a row in the target table to be updated (or inserted in the case of a `MERGE`).

MAPPING

Optional. If a mapping is specified, it overrides the default source-to-target column mapping. The default source-to-target mapping is based on a match of column names as defined in the source `COLUMNS` section and the column names of the target `TABLE`. A mapping is specified as either:

`target_column_name``: `*`source_column_name`*

or

`target_column_name``: '`*`expression`*`'`

Where *expression* is any expression that you would specify in the `SELECT` list of a query, such as a constant value, a column reference, an operator invocation, a function call, and so on.

PRELOAD

Optional. Specifies operations to run prior to the load operation. Right now the only preload operation is `TRUNCATE`.

TRUNCATE

Optional. If set to true, `gpload` will remove all rows in the target table prior to loading it. Default is false.

REUSE_TABLES

Optional. If set to true, `gpload` will not drop the external table objects and staging table objects it creates. These objects will be reused for future load operations that use the same load specifications. This improves performance of trickle loads (ongoing small loads to the same target table).

If `LOG_ERRORS: true` is specified, `REUSE_TABLES: true` must be specified to retain the formatting errors in Greenplum Database error logs. If `REUSE_TABLES: true` is not specified, formatting error information is deleted after the `gpload` operation.

If the *external_table_name* exists, the utility uses the existing table. The utility returns an error if the table schema does not match the `OUTPUT` table schema.

STAGING_TABLE

Optional. Specify the name of the temporary external table that is created during a `gpload` operation. The external table is used by `gpfdist`. `REUSE_TABLES: true` must also specified. If `REUSE_TABLES` is false or not specified, `STAGING_TABLE` is ignored. By default, `gpload` creates a temporary external table with a randomly generated name.

If *external_table_name* contains a period (.), `gpload` returns an error. If the table exists, the utility uses the table. The utility returns an error if the existing table schema does not match the `OUTPUT` table schema.

The utility uses the value of *`SCHEMA`* in the `EXTERNAL` section as the schema for *external_table_name*. If the `SCHEMA` value is `%`, the schema for *external_table_name* is the same as the schema of the target table, the schema of *`TABLE`* in the `OUTPUT` section.

If `SCHEMA` is not set, the utility searches for the table (using the schemas in the database `search_path`). If the table is not found, *external_table_name* is created in the default `PUBLIC` schema.

FAST_MATCH

Optional. If set to true, `gpload` only searches the database for matching external table objects when reusing external tables. The utility does not check the external table column names and column types in the catalog table `pg_attribute` to ensure that the table can be used for a `gpload` operation. Set the value to true to improve `gpload` performance when reusing external table objects and the database catalog table `pg_attribute` contains a large number of rows. The utility returns an error and quits if the column definitions are not compatible.

The default value is false, the utility checks the external table definition column names and column types.

`REUSE_TABLES: true` must also specified. If `REUSE_TABLES` is false or not specified and `FAST_MATCH: true` is specified, `gpload` returns a warning message.

SQL

Optional. Defines SQL commands to run before and/or after the load operation. You can specify multiple `BEFORE` and/or `AFTER` commands. List commands in the order of desired execution.

BEFORE

Optional. An SQL command to run before the load operation starts. Enclose commands in quotes.

AFTER

Optional. An SQL command to run after the load operation completes. Enclose commands in quotes.

Log File Format

Log files output by `gpload` have the following format:

```
timestamp|level|message
```

Where *timestamp* takes the form: `YYYY-MM-DD HH:MM:SS`, *level* is one of `DEBUG`, `LOG`, `INFO`, `ERROR`, and *message* is a normal text message.

Some `INFO` messages that may be of interest in the log files are (where # corresponds to the actual number of seconds, units of data, or failed rows):

```
INFO|running time: #.## seconds
INFO|transferred #.# kB of #.# kB.
INFO|gpload succeeded
INFO|gpload succeeded with warnings
INFO|gpload failed
INFO|1 bad row
INFO|# bad rows
```

Notes

If your database object names were created using a double-quoted identifier (delimited identifier), you must specify the delimited name within single quotes in the `gpload` control file. For example, if you create a table as follows:

```
CREATE TABLE "MyTable" ("MyColumn" text);
```

Your YAML-formatted `gpload` control file would refer to the above table and column names as follows:

```
- COLUMNS:
   - '"MyColumn"': text
OUTPUT:
   - TABLE: public.'"MyTable"'
```

If the YAML control file contains the `ERROR_TABLE` element that was available in Greenplum Database 4.3.x, `gpload` logs a warning stating that `ERROR_TABLE` is not supported, and load errors are handled as if the `LOG_ERRORS` and `REUSE_TABLE` elements were set to `true`. Rows with formatting errors are logged internally when running in single row error isolation mode.

Examples

Run a load job as defined in `my_load.yml`:

```
gpload -f my_load.yml
```

Example load control file:

```
---
VERSION: 1.0.0.1
DATABASE: ops
USER: gpadmin
HOST: mdw-1
PORT: 5432
GPLOAD:
   INPUT:
    - SOURCE:
         LOCAL_HOSTNAME:
           - etl1-1
           - etl1-2
           - etl1-3
           - etl1-4
         PORT: 8081
         FILE:
           - /var/load/data/*
    - COLUMNS:
           - name: text
           - amount: float4
           - category: text
           - descr: text
           - date: date
    - FORMAT: text
    - DELIMITER: '|'
    - ERROR_LIMIT: 25
    - LOG_ERRORS: true
   OUTPUT:
    - TABLE: payables.expenses
    - MODE: INSERT
   PRELOAD:
    - REUSE_TABLES: true
   SQL:
   - BEFORE: "INSERT INTO audit VALUES('start', current_timestamp)"
   - AFTER: "INSERT INTO audit VALUES('end', current_timestamp)"
```

See Also

gpfdist, *CREATE EXTERNAL TABLE*

gplogfilter

Searches through Greenplum Database log files for specified entries.

Synopsis

```
gplogfilter [timestamp_options] [pattern_options]
     [output_options] [input_options] [input_file]

gplogfilter --help

gplogfilter --version
```

Description

The `gplogfilter` utility can be used to search through a Greenplum Database log file for entries matching the specified criteria. If an input file is not supplied, then `gplogfilter` will use the `$MASTER_DATA_DIRECTORY` environment variable to locate the Greenplum master log file in the standard logging location. To read from standard input, use a dash (`-`) as the input file name. Input files may be compressed using `gzip`. In an input file, a log entry is identified by its timestamp in `YYYY-MM-DD [hh:mm[:ss]]` format.

You can also use `gplogfilter` to search through all segment log files at once by running it through the *gpssh* utility. For example, to display the last three lines of each segment log file:

```
gpssh -f seg_host_file
=> source /usr/local/greenplum-db/greenplum_path.sh
=> gplogfilter -n 3 /gpdata/*/pg_log/gpdb*.csv
```

By default, the output of `gplogfilter` is sent to standard output. Use the `-o` option to send the output to a file or a directory. If you supply an output file name ending in `.gz`, the output file will be compressed by default using maximum compression. If the output destination is a directory, the output file is given the same name as the input file.

Options

Timestamp Options

`-b` *`datetime`* | `--begin=`*`datetime`*

Specifies a starting date and time to begin searching for log entries in the format of `YYYY-MM-DD [hh:mm[:ss]]`.

If a time is specified, the date and time must be enclosed in either single or double quotes. This example encloses the date and time in single quotes:

```
gplogfilter -b '2013-05-23 14:33'
```

`-e` *`datetime`* | `--end=`*`datetime`*

Specifies an ending date and time to stop searching for log entries in the format of `YYYY-MM-DD [hh:mm[:ss]]`.

If a time is specified, the date and time must be enclosed in either single or double quotes. This example encloses the date and time in single quotes:

```
gplogfilter -e '2013-05-23 14:33'
```

`-d` *`time`* | `--duration=`*`time`*

Specifies a time duration to search for log entries in the format of `[hh][:mm[:ss]]`. If used without either the `-b` or `-e` option, will use the current time as a basis.

Pattern Matching Options

`-c i [gnore] | r [espect] | --case=i [gnore] | r [espect]`

Matching of alphabetic characters is case sensitive by default unless proceeded by the `--case=ignore` option.

`-C '`*`string`*`' | --columns='`*`string`*`'`

Selects specific columns from the log file. Specify the desired columns as a comma-delimited string of column numbers beginning with 1, where the second column from left is 2, the third is 3, and so on. See "Viewing the Database Server Log Files" in the *Greenplum Database Administrator Guide* for details about the log file format and for a list of the available columns and their associated number.

`-f '`*`string`*`' | --find='`*`string`*`'`

Finds the log entries containing the specified string.

-F '*string*' | --nofind='*string*'

Rejects the log entries containing the specified string.

-m *regex* | --match=*regex*

Finds log entries that match the specified Python regular expression. See *https://docs.python.org/library/re.html* for Python regular expression syntax.

-M *regex* | --nomatch=*regex*

Rejects log entries that match the specified Python regular expression. See *https://docs.python.org/library/re.html* for Python regular expression syntax.

-t | --trouble

Finds only the log entries that have `ERROR:`, `FATAL:`, or `PANIC:` in the first line.

Output Options

-n *integer* | --tail=*integer*

Limits the output to the last *integer* of qualifying log entries found.

-s *offset* [limit] | --slice=*offset* [limit]

From the list of qualifying log entries, returns the *limit* number of entries starting at the *offset* entry number, where an *offset* of zero (`0`) denotes the first entry in the result set and an *offset* of any number greater than zero counts back from the end of the result set.

-o *output_file* | --out=*output_file*

Writes the output to the specified file or directory location instead of `STDOUT`.

-z 0-9 | --zip=0-9

Compresses the output file to the specified compression level using `gzip`, where `0` is no compression and `9` is maximum compression. If you supply an output file name ending in `.gz`, the output file will be compressed by default using maximum compression.

-a | --append

If the output file already exists, appends to the file instead of overwriting it.

Input Options

input_file

The name of the input log file(s) to search through. If an input file is not supplied, `gplogfilter` will use the `$MASTER_DATA_DIRECTORY` environment variable to locate the Greenplum Database master log file. To read from standard input, use a dash (`-`) as the input file name.

-u | --unzip

Uncompress the input file using `gunzip`. If the input file name ends in `.gz`, it will be uncompressed by default.

--help

Displays the online help.

--version

Displays the version of this utility.

Examples

Display the last three error messages in the master log file:

```
gplogfilter -t -n 3
```

Display all log messages in the master log file timestamped in the last 10 minutes:

```
gplogfilter -d :10
```

Display log messages in the master log file containing the string `|con6 cmd11|`:

```
gplogfilter -f '|con6 cmd11|'
```

Using *gpssh*, run `gplogfilter` on the segment hosts and search for log messages in the segment log files containing the string `con6` and save output to a file.

```
gpssh -f seg_hosts_file -e 'source
/usr/local/greenplum-db/greenplum_path.sh ; gplogfilter -f
con6 /gpdata/*/pg_log/gpdb*.csv' > seglog.out
```

See Also

gpssh, *gpscp*

gpmapreduce

Runs Greenplum MapReduce jobs as defined in a YAML specification document.

Synopsis

```
gpmapreduce -f config.yaml [dbname [username]]
     [-k name=value | --key name=value]
     [-h hostname | --host hostname] [-p port| --port port]
     [-U username | --username username] [-W] [-v]

gpmapreduce -x | --explain

gpmapreduce -X | --explain-analyze

gpmapreduce -V | --version

gpmapreduce -h | --help
```

Requirements

The following are required prior to running this program:

- You must have your MapReduce job defined in a YAML file. See *gpmapreduce.yaml* for more information about the format of, and keywords supported in, the Greenplum MapReduce YAML configuration file.
- You must be a Greenplum Database superuser to run MapReduce jobs written in untrusted Perl or Python.
- You must be a Greenplum Database superuser to run MapReduce jobs with `EXEC` and `FILE` inputs.
- You must be a Greenplum Database superuser to run MapReduce jobs with `GPFDIST` input unless the user has the appropriate rights granted.

Description

MapReduce is a programming model developed by Google for processing and generating large data sets on an array of commodity servers. Greenplum MapReduce allows programmers who are familiar with the MapReduce paradigm to write map and reduce functions and submit them to the Greenplum Database parallel engine for processing.

`gpmapreduce` is the Greenplum MapReduce program. You configure a Greenplum MapReduce job via a YAML-formatted configuration file that you pass to the program for execution by the Greenplum Database parallel engine. The Greenplum Database system distributes the input data, executes the program across a set of machines, handles machine failures, and manages the required inter-machine communication.

Options

-f *config.yaml*

Required. The YAML file that contains the Greenplum MapReduce job definitions. Refer to *gpmapreduce.yaml* for the format and content of the parameters that you specify in this file.

-? | --help

Show help, then exit.

-V | --version

Show version information, then exit.

-v | --verbose

Show verbose output.

-x | --explain

Do not run MapReduce jobs, but produce explain plans.

-X | --explain-analyze

Run MapReduce jobs and produce explain-analyze plans.

-k | --keyname=*value*

Sets a YAML variable. A value is required. Defaults to "key" if no variable name is specified.

Connection Options

-h *host* | --host *host*

Specifies the host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

-p *port* | --port *port*

Specifies the TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system user name.

-W | --password

Force a password prompt.

Examples

Run a MapReduce job as defined in `my_mrjob.yaml` and connect to the database `mydatabase`:

```
gpmapreduce -f my_mrjob.yaml mydatabase
```

See Also

gpmapreduce.yaml

gpmapreduce.yaml

gpmapreduce configuration file.

Synopsis

```
%YAML 1.1
---
VERSION: 1.0.0.2
DATABASE: dbname
USER: db_username
HOST: master_hostname
PORT: master_port
```

```
DEFINE:
  - INPUT:
     NAME: input_name
     FILE:
       - hostname:/path/to/file
     GPFDIST:
       - hostname:port/file_pattern
     TABLE: table_name
     QUERY: SELECT_statement
     EXEC: command_string
     COLUMNS:
       - field_name data_type
     FORMAT: TEXT | CSV
     DELIMITER: delimiter_character
     ESCAPE: escape_character
     NULL: null_string
     QUOTE: csv_quote_character
     ERROR_LIMIT: integer
     ENCODING: database_encoding
```

```
  - OUTPUT:
     NAME: output_name
     FILE: file_path_on_client
     TABLE: table_name
     KEYS:
       - column_name
     MODE: REPLACE | APPEND
```

```
  - MAP:
     NAME: function_name
     FUNCTION: function_definition
     LANGUAGE: perl | python | c
     LIBRARY: /path/filename.so
     PARAMETERS:
       - nametype
     RETURNS:
       - nametype
     OPTIMIZE: STRICT IMMUTABLE
     MODE: SINGLE | MULTI
```

```
  - TRANSITION | CONSOLIDATE | FINALIZE:
     NAME: function_name
     FUNCTION: function_definition
     LANGUAGE: perl | python | c
     LIBRARY: /path/filename.so
     PARAMETERS:
       - nametype
     RETURNS:
       - nametype
     OPTIMIZE: STRICT IMMUTABLE
```

```
    MODE: SINGLE | MULTI
```

```
  - REDUCE:
     NAME: reduce_job_name
     TRANSITION: transition_function_name
     CONSOLIDATE: consolidate_function_name
     FINALIZE: finalize_function_name
     INITIALIZE: value
     KEYS:
       - key_name
```

```
  - TASK:
     NAME: task_name
     SOURCE: input_name
     MAP: map_function_name
     REDUCE: reduce_function_name
EXECUTE:
```

```
  - RUN:
     SOURCE: input_or_task_name
     TARGET: output_name
     MAP: map_function_name
     REDUCE: reduce_function_name...
```

Description

You specify the input, map and reduce tasks, and the output for the Greenplum MapReduce `gpmapreduce` program in a YAML-formatted configuration file. (This reference page uses the name `gpmapreduce.yaml` when referring to this file; you may choose your own name for the file.)

The `gpmapreduce` utility processes the YAML configuration file in order, using indentation (spaces) to determine the document hierarchy and the relationships between the sections. The use of white space in the file is significant.

Keys and Values

VERSION

Required. The version of the Greenplum MapReduce YAML specification. Current supported versions are 1.0.0.1, 1.0.0.2, and 1.0.0.3.

DATABASE

Optional. Specifies which database in Greenplum to connect to. If not specified, defaults to the default database or `$PGDATABASE` if set.

USER

Optional. Specifies which database role to use to connect. If not specified, defaults to the current user or `$PGUSER` if set. You must be a Greenplum superuser to run functions written in untrusted Python and Perl. Regular database users can run functions written in trusted Perl. You also must be a database superuser to run MapReduce jobs that contain *FILE*, *GPFDIST* and *EXEC* input types.

HOST

Optional. Specifies Greenplum master host name. If not specified, defaults to localhost or `$PGHOST` if set.

PORT

Optional. Specifies Greenplum master port. If not specified, defaults to 5432 or `$PGPORT` if set.

DEFINE

Required. A sequence of definitions for this MapReduce document. The `DEFINE` section must have at least one `INPUT` definition.

INPUT

Required. Defines the input data. Every MapReduce document must have at least one input defined. Multiple input definitions are allowed in a document, but each input definition can specify only one of these access types: a file, a `gpfdist` file reference, a table in the database, an SQL command, or an operating system command. See *gpfdist* for information about this reference.

NAME

A name for this input. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, task, reduce function and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

FILE

A sequence of one or more input files in the format: `seghostname:/path/to/filename`. You must be a Greenplum Database superuser to run MapReduce jobs with `FILE` input. The file must reside on a Greenplum segment host.

GPFDIST

A sequence identifying one or more running `gpfdist` file servers in the format: `hostname[:port]/file_pattern`. You must be a Greenplum Database superuser to run MapReduce jobs with `GPFDIST` input.

TABLE

The name of an existing table in the database.

QUERY

A SQL `SELECT` command to run within the database.

EXEC

An operating system command to run on the Greenplum segment hosts. The command is run by all segment instances in the system by default. For example, if you have four segment instances per segment host, the command will be run four times on each host. You must be a Greenplum Database superuser to run MapReduce jobs with `EXEC` input.

COLUMNS

Optional. Columns are specified as: `column_name [data_type]`. If not specified, the default is `value text`. The *DELIMITER* character is what separates two data value fields (columns). A row is determined by a line feed character (`0x0a`).

FORMAT

Optional. Specifies the format of the data - either delimited text (`TEXT`) or comma separated values (`CSV`) format. If the data format is not specified, defaults to `TEXT`.

DELIMITER

Optional for *FILE*, *GPFDIST* and *EXEC* inputs. Specifies a single character that separates data values. The default is a tab character in `TEXT` mode, a comma in `CSV` mode. The delimiter character must only appear between any two data value fields. Do not place a delimiter at the beginning or end of a row.

ESCAPE

Optional for *FILE*, *GPFDIST* and *EXEC* inputs. Specifies the single character that is used for C escape sequences (such as `\n`,`\t`,`\100`, and so on) and for escaping data characters that might otherwise be taken as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. The

default escape character is a \ (backslash) for text-formatted files and a " (double quote) for csv-formatted files, however it is possible to specify another character to represent an escape. It is also possible to disable escaping by specifying the value `'OFF'` as the escape value. This is very useful for data such as text-formatted web log data that has many embedded backslashes that are not intended to be escapes.

NULL

Optional for *FILE*, *GPFDIST* and *EXEC* inputs. Specifies the string that represents a null value. The default is `\N` in `TEXT` format, and an empty value with no quotations in `CSV` format. You might prefer an empty string even in `TEXT` mode for cases where you do not want to distinguish nulls from empty strings. Any input data item that matches this string will be considered a null value.

QUOTE

Optional for *FILE*, *GPFDIST* and *EXEC* inputs. Specifies the quotation character for `CSV` formatted files. The default is a double quote ("). In `CSV` formatted files, data value fields must be enclosed in double quotes if they contain any commas or embedded new lines. Fields that contain double quote characters must be surrounded by double quotes, and the embedded double quotes must each be represented by a pair of consecutive double quotes. It is important to always open and close quotes correctly in order for data rows to be parsed correctly.

ERROR_LIMIT

If the input rows have format errors they will be discarded provided that the error limit count is not reached on any Greenplum segment instance during input processing. If the error limit is not reached, all good rows will be processed and any error rows discarded.

ENCODING

Character set encoding to use for the data. Specify a string constant (such as `'SQL_ASCII'`), an integer encoding number, or `DEFAULT` to use the default client encoding. See *Character Set Support* for more information.

OUTPUT

Optional. Defines where to output the formatted data of this MapReduce job. If output is not defined, the default is `STDOUT` (standard output of the client). You can send output to a file on the client host or to an existing table in the database.

NAME

A name for this output. The default output name is `STDOUT`. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, task, reduce function and input names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

FILE

Specifies a file location on the MapReduce client machine to output data in the format: `/path/to/filename`.

TABLE

Specifies the name of a table in the database to output data. If this table does not exist prior to running the MapReduce job, it will be created using the distribution policy specified with *KEYS*.

KEYS

Optional for *TABLE* output. Specifies the column(s) to use as the Greenplum Database distribution key. If the *EXECUTE* task contains a *REDUCE* definition, then the `REDUCE` keys will be used as the table distribution key by default. Otherwise, the first column of the table will be used as the distribution key.

MODE

Optional for *TABLE* output. If not specified, the default is to create the table if it does not already exist, but error out if it does exist. Declaring `APPEND` adds output data to an existing table (provided the table schema matches the output format) without removing any existing data. Declaring `REPLACE` will drop the table if it exists and then recreate it. Both `APPEND` and `REPLACE` will create a new table if one does not exist.

`MAP`

Required. Each `MAP` function takes data structured in (`key`, `value`) pairs, processes each pair, and generates zero or more output (`key`, `value`) pairs. The Greenplum MapReduce framework then collects all pairs with the same key from all output lists and groups them together. This output is then passed to the *REDUCE* task, which is comprised of *TRANSITION | CONSOLIDATE | FINALIZE* functions. There is one predefined `MAP` function named `IDENTITY` that returns (`key`, `value`) pairs unchanged. Although (`key`, `value`) are the default parameters, you can specify other prototypes as needed.

`TRANSITION | CONSOLIDATE | FINALIZE`

`TRANSITION`, `CONSOLIDATE` and `FINALIZE` are all component pieces of *REDUCE*. A `TRANSITION` function is required. `CONSOLIDATE` and `FINALIZE` functions are optional. By default, all take `state` as the first of their input *PARAMETERS*, but other prototypes can be defined as well.

A `TRANSITION` function iterates through each value of a given key and accumulates values in a `state` variable. When the transition function is called on the first value of a key, the `state` is set to the value specified by *INITIALIZE* of a *REDUCE* job (or the default state value for the data type). A transition takes two arguments as input; the current state of the key reduction, and the next value, which then produces a new `state`.

If a `CONSOLIDATE` function is specified, `TRANSITION` processing is performed at the segment-level before redistributing the keys across the Greenplum interconnect for final aggregation (two-phase aggregation). Only the resulting `state` value for a given key is redistributed, resulting in lower interconnect traffic and greater parallelism. `CONSOLIDATE` is handled like a `TRANSITION`, except that instead of `(state + value) => state`, it is `(state + state) => state`.

If a `FINALIZE` function is specified, it takes the final `state` produced by `CONSOLIDATE` (if present) or `TRANSITION` and does any final processing before emitting the final result. `TRANSITION` and `CONSOLIDATE` functions cannot return a set of values. If you need a `REDUCE` job to return a set, then a `FINALIZE` is necessary to transform the final state into a set of output values.

`NAME`

Required. A name for the function. Names must be unique with regards to the names of other objects in this MapReduce job (such as function, task, input and output names). You can also specify the name of a function built-in to Greenplum Database. If using a built-in function, do not supply *LANGUAGE* or a *FUNCTION* body.

`FUNCTION`

Optional. Specifies the full body of the function using the specified *LANGUAGE*. If `FUNCTION` is not specified, then a built-in database function corresponding to *NAME* is used.

`LANGUAGE`

Required when *FUNCTION* is used. Specifies the implementation language used to interpret the function. This release has language support for `perl`, `python`, and `C`. If calling a built-in database function, `LANGUAGE` should not be specified.

`LIBRARY`

Required when *LANGUAGE* is C (not allowed for other language functions). To use this attribute, *VERSION* must be 1.0.0.2. The specified library file must be installed prior

to running the MapReduce job, and it must exist in the same file system location on all Greenplum hosts (master and segments).

PARAMETERS

Optional. Function input parameters. The default type is `text`.

`MAP` default - `key text`, `value text`

`TRANSITION` default - `state text`, `value text`

`CONSOLIDATE` default - `state1 text`, `state2 text` (must have exactly two input parameters of the same data type)

`FINALIZE` default - `state text` (single parameter only)

RETURNS

Optional. The default return type is `text`.

`MAP` default - `key text`, `value text`

`TRANSITION` default - `state text` (single return value only)

`CONSOLIDATE` default - `state text` (single return value only)

`FINALIZE` default - `value text`

OPTIMIZE

Optional optimization parameters for the function:

`STRICT` - function is not affected by `NULL` values

`IMMUTABLE` - function will always return the same value for a given input

MODE

Optional. Specifies the number of rows returned by the function.

`MULTI` - returns 0 or more rows per input record. The return value of the function must be an array of rows to return, or the function must be written as an iterator using `yield` in Python or `return_next` in Perl. `MULTI` is the default mode for `MAP` and `FINALIZE` functions.

`SINGLE` - returns exactly one row per input record. `SINGLE` is the only mode supported for `TRANSITION` and `CONSOLIDATE` functions. When used with `MAP` and `FINALIZE` functions, `SINGLE` mode can provide modest performance improvement.

REDUCE

Required. A `REDUCE` definition names the *TRANSITION | CONSOLIDATE | FINALIZE* functions that comprise the reduction of (`key`, `value`) pairs to the final result set. There are also several predefined `REDUCE` jobs you can execute, which all operate over a column named `value`:

`IDENTITY` - returns (key, value) pairs unchanged

`SUM` - calculates the sum of numeric data

`AVG` - calculates the average of numeric data

`COUNT` - calculates the count of input data

`MIN` - calculates minimum value of numeric data

`MAX` - calculates maximum value of numeric data

NAME

Required. The name of this `REDUCE` job. Names must be unique with regards to the names of other objects in this MapReduce job (function, task, input and output names). Also,

names cannot conflict with existing objects in the database (such as tables, functions or views).

TRANSITION

Required. The name of the `TRANSITION` function.

CONSOLIDATE

Optional. The name of the `CONSOLIDATE` function.

FINALIZE

Optional. The name of the `FINALIZE` function.

INITIALIZE

Optional for `text` and `float` data types. Required for all other data types. The default value for text is `''` . The default value for float is `0.0` . Sets the initial `state` value of the `TRANSITION` function.

KEYS

Optional. Defaults to `[key, *]`. When using a multi-column reduce it may be necessary to specify which columns are key columns and which columns are value columns. By default, any input columns that are not passed to the `TRANSITION` function are key columns, and a column named `key` is always a key column even if it is passed to the `TRANSITION` function. The special indicator `*` indicates all columns not passed to the `TRANSITION` function. If this indicator is not present in the list of keys then any unmatched columns are discarded.

TASK

Optional. A `TASK` defines a complete end-to-end `INPUT`/`MAP`/`REDUCE` stage within a Greenplum MapReduce job pipeline. It is similar to *EXECUTE* except it is not immediately executed. A task object can be called as *INPUT* to further processing stages.

NAME

Required. The name of this task. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, reduce function, input and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

SOURCE

The name of an *INPUT* or another `TASK`.

MAP

Optional. The name of a *MAP* function. If not specified, defaults to `IDENTITY`.

REDUCE

Optional. The name of a *REDUCE* function. If not specified, defaults to `IDENTITY`.

EXECUTE

Required. `EXECUTE` defines the final `INPUT`/`MAP`/`REDUCE` stage within a Greenplum MapReduce job pipeline.

RUN

SOURCE

Required. The name of an *INPUT* or *TASK*.

TARGET

Optional. The name of an *OUTPUT*. The default output is `STDOUT`.

MAP

Optional. The name of a *MAP* function. If not specified, defaults to `IDENTITY`.

REDUCE

Optional. The name of a *REDUCE* function. Defaults to `IDENTITY`.

See Also

gpmapreduce

gpmovemirrors

Moves mirror segment instances to new locations.

Synopsis

```
gpmovemirrors -i move_config_file [-d master_data_directory]
          [-l logfile_directory]
          [-B parallel_processes] [-v]

gpmovemirrors -?

gpmovemirrors --version
```

Description

The `gpmovemirrors` utility moves mirror segment instances to new locations. You may want to move mirrors to new locations to optimize distribution or data storage.

Before moving segments, the utility verifies that they are mirrors, and that their corresponding primary segments are up and are in synchronizing or resynchronizing mode.

By default, the utility will prompt you for the file system location(s) where it will move the mirror segment data directories.

You must make sure that the user who runs `gpmovemirrors` (the `gpadmin` user) has permissions to write to the data directory locations specified. You may want to create these directories on the segment hosts and `chown` them to the appropriate user before running `gpmovemirrors`.

Options

`-B` *`parallel_processes`*

The number of mirror segments to move in parallel. If not specified, the utility will start up to 4 parallel processes depending on how many mirror segment instances it needs to move.

`-d` *`master_data_directory`*

The master data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

`-i` *`move_config_file`*

A configuration file containing information about which mirror segments to move, and where to move them.

You must have one mirror segment listed for each primary segment in the system. Each line inside the configuration file has the following format (as per attributes in the `gp_segment_configuration` catalog table):

```
<old_address>|<port>|<data_dir> <new_address>|<port>|<data_dir>
```

Where `<old_address>` and `<new_address>` are the host names or IP addresses of the segment hosts, `<port>` is the communication port, and `<data_dir>` is the segment instance data directory.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-v (verbose)

Sets logging output to verbose.

--version (show utility version)

Displays the version of this utility.

-? (help)

Displays the online help.

Examples

Moves mirrors from an existing Greenplum Database system to a different set of hosts:

```
$ gpmovemirrors -i move_config_file
```

Where the `move_config_file` looks something like this:

```
sdw2|50000|/data2/mirror/gpseg0 sdw3|50000|/data/mirror/gpseg0
sdw2|50001|/data2/mirror/gpseg1 sdw4|50001|/data/mirror/gpseg1
sdw3|50002|/data2/mirror/gpseg2 sdw1|50002|/data/mirror/gpseg2
```

gppkg

Installs Greenplum Database extensions in `.gppkg` format, such as PL/Java, PL/R, PostGIS, and MADlib, along with their dependencies, across an entire cluster.

Synopsis

```
gppkg [-i package | -u package | -r  name-version | -c]
        [-d master_data_directory] [-a] [-v]

gppkg --migrate GPHOME_old GPHOME_new [-a] [-v]

gppkg [-q | --query] query_option

gppkg -? | --help | -h

gppkg --version
```

Description

The Greenplum Package Manager (`gppkg`) utility installs Greenplum Database extensions, along with any dependencies, on all hosts across a cluster. It will also automatically install extensions on new hosts in the case of system expansion and segment recovery.

> **Note:** After a major upgrade to Greenplum Database, you must download and install all `gppkg` extensions again.

Examples of database extensions and packages software that are delivered using the Greenplum Package Manager are:

- PL/Java
- PL/R
- PostGIS
- MADlib

Options

-a (do not prompt)

Do not prompt the user for confirmation.

-c | --clean

Reconciles the package state of the cluster to match the state of the master host. Running this option after a failed or partial install/uninstall ensures that the package installation state is consistent across the cluster.

-d *master_data_directory*

The master data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-i *package* | --install=*package*

Installs the given package. This includes any pre/post installation steps and installation of any dependencies.

--migrate *GPHOME_old GPHOME_new*

Migrates packages from a separate `$GPHOME`. Carries over packages from one version of Greenplum Database to another.

For example: `gppkg --migrate /usr/local/greenplum-db-<old-version> / usr/local/greenplum-db-<new-version>`

When migrating packages, these requirements must be met.

- At least the master instance of the destination Greenplum Database must be started (the instance installed in *GPHOME_new*). Before running the `gppkg` command start the Greenplum Database master with the command `gpstart -m`.
- Run the `gppkg` utility from the *GPHOME_new* installation. The migration destination installation directory.

-q | --query *query_option*

Provides information specified by `query_option` about the installed packages. Only one `query_option` can be specified at a time. The following table lists the possible values for query_option. `<package_file>` is the name of a package.

Table 65: Query Options for gppkg

query_option	Returns
`<package_file>`	Whether the specified package is installed.
`--info <package_file>`	The name, version, and other information about the specified package.
`--list <package_file>`	The file contents of the specified package.
`--all`	List of all installed packages.

-r *name-version* | --remove=*name-version*

Removes the specified package.

-u *package* | --update=*package*

Updates the given package.

Warning: The process of updating a package includes removing all previous versions of the system objects related to the package. For example, previous versions of shared libraries are removed. After the update process, a database function will fail when it is called if the function references a package file that has been removed.

--version (show utility version)
: Displays the version of this utility.

-v | --verbose
: Sets the logging level to verbose.

-? | -h | --help
: Displays the online help.

gprecoverseg

Recovers a primary or mirror segment instance that has been marked as down (if mirroring is enabled).

Synopsis

```
gprecoverseg [-p new_recover_host[,...]] | -i recover_config_file] [-
d master_data_directory]
             [-B parallel_processes] [-F [-s]] [-a] [-q]
             [--no-progress] [-l logfile_directory]

gprecoverseg -r

gprecoverseg -o output_recover_config_file
             [-p new_recover_host[,...]]

gprecoverseg -? | --help

gprecoverseg --version
```

Description

In a system with mirrors enabled, the `gprecoverseg` utility reactivates a failed segment instance and identifies the changed database files that require resynchronization. Once `gprecoverseg` completes this process, the system goes into *resyncronizing* mode until the recovered segment is brought up to date. The system is online and fully operational during resyncronization.

During an incremental recovery (the `-F` option is not specified), if `gprecoverseg` detects a segment instance with mirroring disabled in a system with mirrors enabled, the utility reports that mirroring is disabled for the segment, does not attempt to recover that segment instance, and continues the recovery process.

A segment instance can fail for several reasons, such as a host failure, network failure, or disk failure. When a segment instance fails, its status is marked as *down* in the Greenplum Database system catalog, and its mirror is activated in *change tracking* mode. In order to bring the failed segment instance back into operation again, you must first correct the problem that made it fail in the first place, and then recover the segment instance in Greenplum Database using `gprecoverseg`.

> **Note:** If incremental recovery was not successful and the down segment instance data is not corrupted, contact Pivotal Support.

Segment recovery using `gprecoverseg` requires that you have an active mirror to recover from. For systems that do not have mirroring enabled, or in the event of a double fault (a primary and mirror pair both down at the same time) — you must take manual steps to recover the failed segment instances and then perform a system restart to bring the segments back online. For example, this command restarts a system.

```
gpstop -r
```

By default, a failed segment is recovered in place, meaning that the system brings the segment back online on the same host and data directory location on which it was originally configured. In this case, use the following format for the recovery configuration file (using `-i`).

```
<failed_host_address>|<port>|<data_directory>
```

In some cases, this may not be possible (for example, if a host was physically damaged and cannot be recovered). In this situation, `gprecoverseg` allows you to recover failed segments to a completely new host (using `-p`), on an alternative data directory location on your remaining live segment hosts (using `-s`), or by supplying a recovery configuration file (using `-i`) in the following format. The word **SPACE** indicates the location of a required space. Do not add additional spaces.

```
<failed_host_address>|<port>|<data_directory>SPACE
<recovery_host_address>|<port>|<data_directory>
```

See the `-i` option below for details and examples of a recovery configuration file.

The `gp_segment_configuration` system catalog table can help you determine your current segment configuration so that you can plan your mirror recovery configuration. For example, run the following query:

```
=# SELECT dbid, content, address, port, datadir
   FROM gp_segment_configuration
   ORDER BY dbid;
```

The new recovery segment host must be pre-installed with the Greenplum Database software and configured exactly the same as the existing segment hosts. A spare data directory location must exist on all currently configured segment hosts and have enough disk space to accommodate the failed segments.

The recovery process marks the segment as up again in the Greenplum Database system catalog, and then initiates the resyncronization process to bring the transactional state of the segment up-to-date with the latest changes. The system is online and available during resyncronization. To check the status of the resyncronization process run:

```
gpstate -m
```

Options

`-a (do not prompt)`

Do not prompt the user for confirmation.

`-B` *`parallel_processes`*

The number of segments to recover in parallel. If not specified, the utility will start up to 16 parallel processes depending on how many segment instances it needs to recover.

`-d` *`master_data_directory`*

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

`-F (full recovery)`

Optional. Perform a full copy of the active segment instance in order to recover the failed segment. The default is to only copy over the incremental changes that occurred while the segment was down.

> **Warning:** A full recovery deletes the data directory of the down segment instance before copying the data from the active (current primary) segment instance. Before performing a full recovery, ensure that the segment failure did not cause data corruption and that any host segment disk issues have been fixed.

Also, for a full recovery, the utility does not restore custom files that are stored in the segment instance data directory even if the custom files are also in the active segment instance. You must restore the custom files manually. For example, when using the `gpfdists` protocol (`gpfdist` with SSL encryption) to manage external data, client certificate files are required in the segment instance `$PGDATA/gpfdists` directory. These files are not restored. For information about configuring `gpfdists`, see *Encrypting gpfdist Connections*.

Full recovery can take a long time for large databases, so `gprecoverseg` displays a running estimate of the completion progress of the copy for each segment. Progress for each segment is updated once per second, using ANSI escape codes to update the line for each segment in-place. If you are redirecting the `gprecoverseg` output to a file, or if the ANSI escape codes do not work correctly on your terminal, you can include the `-s` option on the command line to omit the ANSI escape codes. This outputs a new line once per second for each segment. Include the `--no-progress` option to completely disable the progress reports.

`-i` *`recover_config_file`*

Specifies the name of a file with the details about failed segments to recover. Each line in the file is in the following format. The word **SPACE** indicates the location of a required space. Do not add additional spaces.

```
<failed_host_address>|<port>|<data_directory>SPACE
<recovery_host_address>|<port>|<data_directory>
```

Comments

Lines beginning with # are treated as comments and ignored.

Segments to Recover

Each line after the first specifies a segment to recover. This line can have one of two formats. In the event of in-place recovery, enter one group of colon delimited fields in the line. For example:

```
failedAddress|failedPort|failedDataDirectory
```

For recovery to a new location, enter two groups of fields separated by a space in the line. The required space is indicated by **SPACE**. Do not add additional spaces.

```
failedAddress|failedPort|failedDataDirectorySPACEnewAddress|
newPort|newDataDirectory
```

Examples

In-place recovery of a single mirror

```
sdw1-1|50001|/data1/mirror/gpseg16
```

Recovery of a single mirror to a new host

```
sdw1-1|50001|/data1/mirror/gpseg16SPACEsdw4-1|50001|/data1/
recover1/gpseg16
```

Obtaining a Sample File

You can use the `-o` option to output a sample recovery configuration file to use as a starting point.

`-l` *`logfile_directory`*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-o *output_recover_config_file*

Specifies a file name and location to output a sample recovery configuration file. The output file lists the currently invalid segments and their default recovery location in the format that is required by the `-i` option. Use together with the `-p` option to output a sample file for recovering on a different host. This file can be edited to supply alternate recovery locations if needed.

-p *new_recover_host*[,...]

Specifies a spare host outside of the currently configured Greenplum Database array on which to recover invalid segments. In the case of multiple failed segment hosts, you can specify a comma-separated list. The spare host must have the Greenplum Database software installed and configured, and have the same hardware and OS configuration as the current segment hosts (same OS version, locales, `gpadmin` user account, data directory locations created, ssh keys exchanged, number of network interfaces, network interface naming convention, and so on.).

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-r (rebalance segments)

After a segment recovery, segment instances may not be returned to the preferred role that they were given at system initialization time. This can leave the system in a potentially unbalanced state, as some segment hosts may have more active segments than is optimal for top system performance. This option rebalances primary and mirror segments by returning them to their preferred roles. All segments must be valid and synchronized before running `gprecoverseg -r`. If there are any in progress queries, they will be cancelled and rolled back.

-s

Show `pg_basebackup` progress sequentially instead of in-place. Useful when writing to a file, or if a tty does not support escape sequences. The default is to show progress in-place.

--no-progress

Suppresses progress reports from the `pg_basebackup` utility. The default is to display progress of base backups.

-v (verbose)

Sets logging output to verbose.

--version (version)

Displays the version of this utility.

-? (help)

Displays the online help.

Examples

Recover any failed segment instances in place:

```
$ gprecoverseg
```

Rebalance your Greenplum Database system after a recovery by resetting all segments to their preferred role. First check that all segments are up and synchronized.

```
$ gpstate -m
$ gprecoverseg -r
```

Recover any failed segment instances to a newly configured spare segment host:

```
$ gprecoverseg -i recover_config_file
```

Output the default recovery configuration file:

```
$ gprecoverseg -o /home/gpadmin/recover_config_file
```

See Also

gpstart, *gpstop*

gpreload

Reloads Greenplum Database table data sorting the data based on specified columns.

Synopsis

```
gpreload -d database [-p port] {-t | --table-file} path_to_file [-a]

gpreload -h

gpreload --version
```

Description

The `gpreload` utility reloads table data with column data sorted. For tables that were created with the table storage option `appendoptimized=TRUE` and compression enabled, reloading the data with sorted data can improve table compression. You specify a list of tables to be reloaded and the table columns to be sorted in a text file.

Compression is improved by sorting data when the data in the column has a relatively low number of distinct values when compared to the total number of rows.

For a table being reloaded, the order of the columns to be sorted might affect compression. The columns with the fewest distinct values should be listed first. For example, listing state then city would generally result in better compression than listing city then state.

```
public.cust_table: state, city
public.cust_table: city, state
```

For information about the format of the file used with `gpreload`, see the `--table-file` option.

Notes

To improve reload performance, indexes on tables being reloaded should be removed before reloading the data.

Running the `ANALYZE` command after reloading table data might query performance because of a change in the data distribution of the reloaded data.

For each table, the utility copies table data to a temporary table, truncates the existing table data, and inserts data from the temporary table to the table in the specified sort order. Each table reload is performed in a single transaction.

For a partitioned table, you can reload the data of a leaf child partition. However, data is inserted from the root partition table, which acquires a `ROW EXCLUSIVE` lock on the entire table.

Options

-a (do not prompt)

Optional. If specified, the `gpreload` utility does not prompt the user for confirmation.

-d *database*

The database that contains the tables to be reloaded. The `gpreload` utility connects to the database as the user running the utility.

-p *port*

The Greenplum Database master port. If not specified, the value of the `PGPORT` environment variable is used. If the value is not available, an error is returned.

{-t | --table-file } *path_to_file*

The location and name of file containing list of schema qualified table names to reload and the column names to reorder from the Greenplum Database. Only user defined tables are supported. Views or system catalog tables are not supported.

If indexes are defined on table listed in the file, `gpreload` prompts to continue.

Each line specifies a table name and the list of columns to sort. This is the format of each line in the file:

```
schema.table_name: column [desc] [, column2 [desc] ... ]
```

The table name is followed by a colon (:) and then at least one column name. If you specify more than one column, separate the column names with a comma. The columns are sorted in ascending order. Specify the keyword `desc` after the column name to sort the column in descending order.

Wildcard characters are not supported.

If there are errors in the file, `gpreload` reports the first error and exits. No data is reloaded.

The following example reloads three tables:

```
public.clients: region, state, rep_id desc
public.merchants: region, state
test.lineitem: group, assy, whse
```

In the first table `public.clients`, the data in the `rep_id` column is sorted in descending order. The data in the other columns are sorted in ascending order.

--version (show utility version)

Displays the version of this utility.

-? (help)

Displays the online help.

Example

This example command reloads the tables in the database `mytest` that are listed in the file `data-tables.txt`.

```
gpreload -d mytest --table-file data-tables.txt
```

See Also

`CREATE TABLE` in the *Greenplum Database Reference Guide*

gprestore

Restore a Greenplum Database backup that was created using the `gpbackup` utility. By default `gprestore` uses backed up metadata files and DDL files located in the Greenplum Database master host data directory, with table data stored locally on segment hosts in CSV data files.

Synopsis

```
gprestore --timestamp YYYYMMDDHHMMSS
   [--backup-dir directory]
   [--create-db]
   [--debug]
   [--exclude-schema schema_name [--exclude-schema schema_name ...]]
   [--exclude-table schema.table [--exclude-table schema.table ...]]
   [--exclude-table-file file_name]
   [--exclude-schema-file file_name]
   [--include-schema schema_name [--include-schema schema_name ...]]
   [--include-table schema.table [--include-table schema.table ...]]
   [--include-schema-file file_name]
   [--include-table-file file_name]
   [--redirect-schema schema_name]
   [--data-only | --metadata-only]
   [--incremental]
   [--jobs int]
   [--on-error-continue]
   [--plugin-config config_file_location]
   [--quiet]
   [--redirect-db database_name]
   [--verbose]
   [--version]
   [--with-globals]
   [--with-stats]

gprestore --help
```

Description

To use `gprestore` to restore from a backup set, you must include the `--timestamp` option to specify the exact timestamp value (YYYYMMDDHHMMSS) of the backup set to restore. If you specified a custom `--backup-dir` to consolidate the backup files, include the same `--backup-dir` option with `gprestore` to locate the backup files.

If the backup you specify is an incremental backup, you need a complete set of backup files (a full backup and any required incremental backups). `gprestore` ensures that the complete backup set is available before attempting to restore a backup.

> **Important:** For incremental backup sets, the backups must be on a single device. For example, a backup set must all be on a Data Domain system.

For information about incremental backups, see *Creating and Using Incremental Backups with gpbackup and gprestore*.

When restoring from a backup set, `gprestore` restores to a database with the same name as the name specified when creating the backup set. If the target database exists and a table being restored exists in the database, the restore operation fails. Include the `--create-db` option if the target database does not exist in the cluster. You can optionally restore a backup set to a different database by using the `--redirect-db` option.

When restoring a backup set that contains data from some leaf partitions of a partitioned tables, the partitioned table is restored along with the data for the leaf partitions. For example, you create a backup

with the gpbackup option --include-table-file and the text file lists some leaf partitions of a partitioned table. Restoring the backup creates the partitioned table and restores the data only for the leaf partitions listed in the file.

Greenplum Database system objects are automatically included in a gpbackup backup set, but these objects are only restored if you include the --with-globals option to gprestore. Similarly, if you backed up query plan statistics using the --with-stats option, you can restore those statistics by providing --with-stats to gprestore. By default, only database objects in the backup set are restored.

When a materialized view is restored, the data is not restored. To populate the materialized view with data, use REFRESH MATERIALIZED VIEW. The tables that are referenced by the materialized view definition must be available. The gprestore log file lists the materialized views that were restored and the REFRESH MATERIALIZED VIEW commands that are used to populate the materialized views with data.

Performance of restore operations can be improved by creating multiple parallel connections to restore table data and metadata. By default gprestore uses 1 connection, but you can increase this number with the --jobs option for large restore operations.

When a restore operation completes, gprestore returns a status code. See *Return Codes*.

gprestore can send status email notifications after a back up operation completes. You specify when the utility sends the mail and the email recipients in a configuration file. See *Configuring Email Notifications*.

> **Note:** This utility uses secure shell (SSH) connections between systems to perform its tasks. In large Greenplum Database deployments, cloud deployments, or deployments with a large number of segments per host, this utility may exceed the host's maximum threshold for unauthenticated connections. Consider updating the SSH MaxStartups and MaxSessions configuration parameters to increase this threshold. For more information about SSH configuration options, refer to the SSH documentation for your Linux distribution.

Options

--timestamp *YYYYMMDDHHMMSS*

Required. Specifies the timestamp of the gpbackup backup set to restore. By default gprestore tries to locate metadata files for the timestamp on the Greenplum Database master host in the $MASTER_DATA_DIRECTORY/backups/YYYYMMDD/YYYYMMDDhhmmss/ directory, and CSV data files in the <seg_dir>/backups/YYYYMMDD/YYYYMMDDhhmmss/ directory of each segment host.

--backup-dir *directory*

Optional. Sources all backup files (metadata files and data files) from the specified directory. You must specify *directory* as an absolute path (not relative). If you do not supply this option, gprestore tries to locate metadata files for the timestamp on the Greenplum Database master host in the $MASTER_DATA_DIRECTORY/backups/YYYYMMDD/YYYYMMDDhhmmss/ directory. CSV data files must be available on each segment in the <seg_dir>/backups/YYYYMMDD/YYYYMMDDhhmmss/ directory. Include this option when you specify a custom backup directory with gpbackup.

You cannot combine this option with the option --plugin-config.

--create-db

Optional. Creates the database before restoring the database object metadata.

The database is created by cloning the empty standard system database template0.

--data-only

Optional. Restores table data from a backup created with the gpbackup utility, without creating the database tables. This option assumes the tables exist in the target database. To restore data for a specific set of tables from a backup set, you can specify an option to include tables or schemas or exclude tables or schemas. Specify the --with-stats option to restore table statistics from the backup.

The backup set must contain the table data to be restored. For example, a backup created with the `gpbackup` option `--metadata-only` does not contain table data.

To restore only database tables, without restoring the table data, see the option `--metadata-only`.

`--debug`

Optional. Displays verbose and debug log messages during a restore operation.

`--exclude-schema *schema_name*`

Optional. Specifies a database schema to exclude from the restore operation. You can specify this option multiple times. You cannot combine this option with the option `--include-schema`, `--include-schema-file`, or a table filtering option such as `--include-table`.

`--exclude-schema-file *file_name*`

Optional. Specifies a text file containing a list of schemas to exclude from the backup. Each line in the text file must define a single schema. The file must not include trailing lines. If a schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. You cannot combine this option with the option `--include-schema` or `--include-schema-file`, or a table filtering option such as `--include-table`.

`--exclude-table *schema.table*`

Optional. Specifies a table to exclude from the restore operation. You can specify this option multiple times. The table must be in the format `<schema-name>.<table-name>`. If a table or schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. You can specify this option multiple times. If the table is not in the backup set, the restore operation fails. You cannot specify a leaf partition of a partitioned table.

You cannot combine this option with the option `--exclude-schema`, `--exclude-schema-file`, or another a table filtering option such as `--include-table`.

`--exclude-table-file *file_name*`

Optional. Specifies a text file containing a list of tables to exclude from the restore operation. Each line in the text file must define a single table using the format `<schema-name>.<table-name>`. The file must not include trailing lines. If a table or schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. If a table is not in the backup set, the restore operation fails. You cannot specify a leaf partition of a partitioned table.

You cannot combine this option with the option `--exclude-schema`, `--exclude-schema-file`, or another a table filtering option such as `--include-table`.

`--include-schema *schema_name*`

Optional. Specifies a database schema to restore. You can specify this option multiple times. If you specify this option, any schemas that you specify must be available in the backup set. Any schemas that are not included in subsequent `--include-schema` options are omitted from the restore operation.

If a schema that you specify for inclusion exists in the database, the utility issues an error and continues the operation. The utility fails if a table being restored exists in the database.

You cannot use this option if objects in the backup set have dependencies on multiple schemas.

See *Filtering the Contents of a Backup or Restore* for more information.

`--include-schema-file *file_name*`

Optional. Specifies a text file containing a list of schemas to restore. Each line in the text file must define a single schema. The file must not include trailing lines. If a schema name

uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes.

The schemas must exist in the backup set. Any schemas not listed in this file are omitted from the restore operation.

You cannot use this option if objects in the backup set have dependencies on multiple schemas.

`--include-table` *`schema.table`*

Optional. Specifies a table to restore. The table must be in the format `<schema-name>.<table-name>`. If a table or schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. You can specify this option multiple times. You cannot specify a leaf partition of a partitioned table.

You can also specify the qualified name of a sequence, a view, or a materialized view.

If you specify this option, the utility does not automatically restore dependent objects. You must also explicitly specify the dependent objects that are required. For example if you restore a view or a materialized view, you must also restore the tables that the view or the materialized view uses. If you restore a table that uses a sequence, you must also restore the sequence. The dependent objects must exist in the backup set.

You cannot combine this option with a schema filtering option such as `--include-schema`, or another table filtering option such as `--exclude-table-file`.

`--include-table-file` *`file_name`*

Optional. Specifies a text file containing a list of tables to restore. Each line in the text file must define a single table using the format `<schema-name>.<table-name>`. The file must not include trailing lines. If a table or schema name uses any character other than a lowercase letter, number, or an underscore character, then you must include that name in double quotes. Any tables not listed in this file are omitted from the restore operation. You cannot specify a leaf partition of a partitioned table.

You can also specify the qualified name of a sequence, a view, or a materialized view.

If you specify this option, the utility does not automatically restore dependent objects. You must also explicitly specify dependent objects that are required. For example if you restore a view or a materialized view, you must also specify the tables that the view or the materialized uses. If you specify a table that uses a sequence, you must also specify the sequence. The dependent objects must exist in the backup set.

For a materialized view, the data is not restored. To populate the materialized view with data, you must use `REFRESH MATERIALIZED VIEW` and the tables that are referenced by the materialized view definition must be available.

If you use the `--include-table-file` option, `gprestore` does not create roles or set the owner of the tables. The utility restores table indexes and rules. Triggers are also restored but are not supported in Greenplum Database.

See *Filtering the Contents of a Backup or Restore* for more information.

`--incremental (Beta)`

Restores only the table data in the incremental backup specified by the `--timestamp` option. Table data is not restored from previous incremental backups in the backup set. For information about incremental backups, see *Creating and Using Incremental Backups with gpbackup and gprestore*.

> **Warning:** This is a Beta feature and is not supported in a production environment.

An incremental backup contains the following table data that can be restored.

- Data from all heap tables.
- Data from append-optimized tables that have been modified since the previous backup.
- Data from leaf partitions that have been modified from the previous backup.

When this option is specified, `gprestore` restores table data by truncating the table and reloading data into the table.

> **Warning:** When this option is specified, `gpbackup` assumes that no changes have been made to the table definitions of the tables being restored, such as adding or removing columns.

`--redirect-schema` *`schema_name`*

Optional. Restore data in the specified schema instead of the original schemas. The specified schema must already exist. If the data being restored is in multiple schemas, all the data is redirected into the specified schema.

This option must be used with an option that includes tables, `--inlcude-table` or `--include table-file`.

You cannot use this option with an option that excludes schemas or tables such as `--exclude-schema` or `--exclude-table`.

You can use this option with the `--metadata-only` or `--data-only` options.

`--jobs` *`int`*

Optional. Specifies the number of parallel connections to use when restoring table data and metadata. By default, `gprestore` uses 1 connection. Increasing this number can improve the speed of restoring data.

> **Note:** If you used the `gpbackup --single-data-file` option to combine table backups into a single file per segment, you cannot set `--jobs` to a value higher than 1 to perform a parallel restore operation.

`--metadata-only`

Optional. Creates database tables from a backup created with the `gpbackup` utility, but does not restore the table data. This option assumes the tables do not exist in the target database. To create a specific set of tables from a backup set, you can specify an option to include tables or schemas or exclude tables or schemas. Specify the option `--with-globals` to restore the Greenplum Database system objects.

The backup set must contain the DDL for tables to be restored. For example, a backup created with the `gpbackup` option `--data-only` does not contain the DDL for tables.

To restore table data after you create the database tables, see the option *`--data-only`*.

`--on-error-continue`

Optional. Specify this option to continue the restore operation if an SQL error occurs when creating database metadata (such as tables, roles, or functions) or restoring data. If another type of error occurs, the utility exits. The default is to exit on the first error.

When this option is included, the utility displays an error summary and writes error information to the `gprestore` log file and continues the restore operation. The utility also creates text files in the backup directory that contain the list of tables that generated SQL errors.

- Tables with metadata errors - `gprestore_<backup-timestamp>_<restore-time>_error_tables_metadata`
- Tables with data errors - `gprestore_<backup-timestamp>_<restore-time>_error_tables_data`

`--plugin-config` *`config-file_location`*

Specify the location of the `gpbackup` plugin configuration file, a YAML-formatted text file. The file contains configuration information for the plugin application that `gprestore` uses during the restore operation.

If you specify the `--plugin-config` option when you back up a database, you must specify this option with configuration information for a corresponding plugin application when you restore the database from the backup.

You cannot combine this option with the option `--backup-dir`.

For information about using storage plugin applications, see *Using gpbackup Storage Plugins*.

`--quiet`

Optional. Suppress all non-warning, non-error log messages.

`--redirect-db` *`database_name`*

Optional. Restore to the specified *database_name* instead of to the database that was backed up.

`--verbose`

Optional. Displays verbose log messages during a restore operation.

`--version`

Optional. Print the version number and exit.

`--with-globals`

Optional. Restores Greenplum Database system objects in the backup set, in addition to database objects. See *Objects Included in a Backup or Restore*.

`--with-stats`

Optional. Restore query plan statistics from the backup set.

`--help`

Displays the online help.

Return Codes

One of these codes is returned after `gprestore` completes.

- **0** – Restore completed with no problems.
- **1** – Restore completed with non-fatal errors. See log file for more information.
- **2** – Restore failed with a fatal error. See log file for more information.

Examples

Create the demo database and restore all schemas and tables in the backup set for the indicated timestamp:

```
$ dropdb demo
$ gprestore --timestamp 20171103152558 --create-db
```

Restore the backup set to the "demo2" database instead of the "demo" database that was backed up:

```
$ createdb demo2
$ gprestore --timestamp 20171103152558 --redirect-db demo2
```

Restore global Greenplum Database metadata and query plan statistics in addition to the database objects:

```
$ gprestore --timestamp 20171103152558 --create-db --with-globals --with-
stats
```

Restore, using backup files that were created in the `/home/gpadmin/backup` directory, creating 8 parallel connections:

```
$ gprestore --backup-dir /home/gpadmin/backups/ --timestamp 20171103153156
 --create-db --jobs 8
```

Restore only the "wikipedia" schema included in the backup set:

```
$ dropdb demo
$ gprestore --include-schema wikipedia --backup-dir /home/gpadmin/backups/
 --timestamp 20171103153156 --create-db
```

If you restore from an incremental backup set, all the required files in the backup set must be available to `gprestore`. For example, the following timestamp keys specify an incremental backup set. 20170514054532 is the full backup and the others are incremental backups.

```
20170514054532 (full backup)
20170714095512
20170914081205
20171114064330
20180114051246
```

The following `gprestore` command specifies the timestamp 20121114064330. The incremental backup with the timestamps 20120714095512 and 20120914081205 and the full backup must be available to perform a restore.

```
gprestore --timestamp 20121114064330 --redirect-db mystest --create-db
```

See Also

gpbackup, *Parallel Backup with gpbackup and gprestore* and *Using the S3 Storage Plugin with gpbackup and gprestore*

gpscp

Copies files between multiple hosts at once.

Synopsis

```
gpscp { -f hostfile_gpssh | -h hostname [-h hostname ...] }
      [-J character] [-v] [[user@]hostname:]file_to_copy [...]
      [[user@]hostname:]copy_to_path

gpscp -?

gpscp --version
```

Description

The `gpscp` utility allows you to copy one or more files from the specified hosts to other specified hosts in one command using SCP (secure copy). For example, you can copy a file from the Greenplum Database master host to all of the segment hosts at the same time.

To specify the hosts involved in the SCP session, use the `-f` option to specify a file containing a list of host names, or use the `-h` option to name single host names on the command-line. At least one host name (`-h`) or a host file (`-f`) is required. The `-J` option allows you to specify a single character to substitute for the *hostname* in the `copy from` and `copy to` destination strings. If `-J` is not specified, the default substitution character is an equal sign (=). For example, the following command will copy `.bashrc` from the local host to `/home/gpadmin` on all hosts named in `hostfile_gpssh`:

```
gpscp -f hostfile_gpssh .bashrc =:/home/gpadmin
```

If a user name is not specified in the host list or with *user*@ in the file path, `gpscp` will copy files as the currently logged in user. To determine the currently logged in user, do a `whoami` command. By default, `gpscp` goes to `$HOME` of the session user on the remote hosts after login. To ensure the file is copied to the correct location on the remote hosts, it is recommended that you use absolute paths.

Before using `gpscp`, you must have a trusted host setup between the hosts involved in the SCP session. You can use the utility `gpssh-exkeys` to update the known host files and exchange public keys between hosts if you have not done so already.

Options

`-f` *`hostfile_gpssh`*

Specifies the name of a file that contains a list of hosts that will participate in this SCP session. The syntax of the host file is one host per line as follows:

```
<hostname>
```

`-h` *`hostname`*

Specifies a single host name that will participate in this SCP session. You can use the `-h` option multiple times to specify multiple host names.

`-J` *`character`*

The `-J` option allows you to specify a single character to substitute for the *hostname* in the `copy from` and `copy to` destination strings. If `-J` is not specified, the default substitution character is an equal sign (=).

`-v (verbose mode)`

Optional. Reports additional messages in addition to the SCP command output.

`file_to_copy`

Required. The file name (or absolute path) of a file that you want to copy to other hosts (or file locations). This can be either a file on the local host or on another named host.

`copy_to_path`

Required. The path where you want the file(s) to be copied on the named hosts. If an absolute path is not used, the file will be copied relative to `$HOME` of the session user. You can also use the equal sign '=' (or another character that you specify with the `-J` option) in place of a *hostname*. This will then substitute in each host name as specified in the supplied host file (`-f`) or with the `-h` option.

`-? (help)`

Displays the online help.

`--version`

Displays the version of this utility.

Examples

Copy the file named `installer.tar` to `/` on all the hosts in the file `hostfile_gpssh`.

```
gpscp -f hostfile_gpssh installer.tar =:/
```

Copy the file named *myfuncs.so* to the specified location on the hosts named `sdw1` and `sdw2`:

```
gpscp -h sdw1 -h sdw2 myfuncs.so =:/usr/local/greenplum-db/lib
```

See Also

gpssh, *gpssh-exkeys*

gpssh

Provides SSH access to multiple hosts at once.

Synopsis

```
gpssh { -f hostfile_gpssh | - h hostname [-h hostname ...] } [-s] [-e]
      [-d seconds] [-t multiplier] [-v]
      [bash_command]

gpssh -?

gpssh --version
```

Description

The `gpssh` utility allows you to run bash shell commands on multiple hosts at once using SSH (secure shell). You can execute a single command by specifying it on the command-line, or omit the command to enter into an interactive command-line session.

To specify the hosts involved in the SSH session, use the `-f` option to specify a file containing a list of host names, or use the `-h` option to name single host names on the command-line. At least one host name (`-h`) or a host file (`-f`) is required. Note that the current host is ***not*** included in the session by default — to include the local host, you must explicitly declare it in the list of hosts involved in the session.

Before using `gpssh`, you must have a trusted host setup between the hosts involved in the SSH session. You can use the utility `gpssh-exkeys` to update the known host files and exchange public keys between hosts if you have not done so already.

If you do not specify a command on the command-line, `gpssh` will go into interactive mode. At the `gpssh` command prompt (=>), you can enter a command as you would in a regular bash terminal command-line, and the command will be executed on all hosts involved in the session. To end an interactive session, press `CTRL+D` on the keyboard or type `exit` or `quit`.

If a user name is not specified in the host file, `gpssh` will execute commands as the currently logged in user. To determine the currently logged in user, do a `whoami` command. By default, `gpssh` goes to `$HOME` of the session user on the remote hosts after login. To ensure commands are executed correctly on all remote hosts, you should always enter absolute paths.

If you encounter network timeout problems when using `gpssh`, you can use `-d` and `-t` options or set parameters in the `gpssh.conf` file to control the timing that `gpssh` uses when validating the initial `ssh` connection. For information about the configuration file, see *gpssh Configuration File*.

Options

bash_command

A bash shell command to execute on all hosts involved in this session (optionally enclosed in quotes). If not specified, `gpssh` starts an interactive session.

`-d (delay)` *`seconds`*

Optional. Specifies the time, in seconds, to wait at the start of a `gpssh` interaction with `ssh`. Default is `0.05`. This option overrides the `delaybeforesend` value that is specified in the `gpssh.conf` configuration file.

Increasing this value can cause a long wait time during `gpssh` startup.

`-e (echo)`

Optional. Echoes the commands passed to each host and their resulting output while running in non-interactive mode.

`-f` *`hostfile_gpssh`*

Specifies the name of a file that contains a list of hosts that will participate in this SSH session. The syntax of the host file is one host per line.

`-h` *`hostname`*

Specifies a single host name that will participate in this SSH session. You can use the `-h` option multiple times to specify multiple host names.

`-s`

Optional. If specified, before executing any commands on the target host, `gpssh` sources the file `greenplum_path.sh` in the directory specified by the `$GPHOME` environment variable.

This option is valid for both interactive mode and single command mode.

`-t` *`multiplier`*

Optional. A decimal number greater than 0 (zero) that is the multiplier for the timeout that `gpssh` uses when validating the `ssh` prompt. Default is `1`. This option overrides the `prompt_validation_timeout` value that is specified in the `gpssh.conf` configuration file.

Increasing this value has a small impact during `gpssh` startup.

`-v (verbose mode)`

Optional. Reports additional messages in addition to the command output when running in non-interactive mode.

`--version`

Displays the version of this utility.

`-? (help)`

Displays the online help.

gpssh Configuration File

The `gpssh.conf` file contains parameters that let you adjust the timing that `gpssh` uses when validating the initial `ssh` connection. These parameters affect the network connection before the `gpssh` session executes commands with `ssh`. The location of the file is specified by the environment variable `MASTER_DATA_DIRECTORY`. If the environment variable is not defined or the `gpssh.conf` file does not exist, `gpssh` uses the default values or the values set with the `-d` and `-t` options. For information about the environment variable, see the *Greenplum Database Reference Guide*.

The `gpssh.conf` file is a text file that consists of a `[gpssh]` section and parameters. On a line, the # (pound sign) indicates the start of a comment. This is an example `gpssh.conf` file.

```
[gpssh]
delaybeforesend = 0.05
prompt_validation_timeout = 1.0
```

```
sync_retries = 5
```

These are the `gpssh.conf` parameters.

`delaybeforesend = seconds`

Specifies the time, in seconds, to wait at the start of a `gpssh` interaction with `ssh`. Default is 0.05. Increasing this value can cause a long wait time during `gpssh` startup. The `-d` option overrides this parameter.

`prompt_validation_timeout = multiplier`

A decimal number greater than 0 (zero) that is the multiplier for the timeout that `gpssh` uses when validating the `ssh` prompt. Increasing this value has a small impact during `gpssh` startup. Default is `1`. The `-t` option overrides this parameter.

`sync_retries = attempts`

A non-negative integer that specifies the maximum number of times that `gpssh` attempts to connect to a remote Greenplum Database host. The default is 3. If the value is 0, `gpssh` returns an error if the initial connection attempt fails. Increasing the number of attempts also increases the time between retry attempts. This parameter cannot be configured with a command-line option.

The `-t` option also affects the time between retry attempts.

Increasing this value can compensate for slow network performance or segment host performance issues such as heavy CPU or I/O load. However, when a connection cannot be established, an increased value also increases the delay when an error is returned.

Examples

Start an interactive group SSH session with all hosts listed in the file `hostfile_gpssh`:

```
$ gpssh -f hostfile_gpssh
```

At the `gpssh` interactive command prompt, run a shell command on all the hosts involved in this session.

```
=> ls -a /data/primary/*
```

Exit an interactive session:

```
=> exit
=> quit
```

Start a non-interactive group SSH session with the hosts named `sdw1` and `sdw2` and pass a file containing several commands named `command_file` to `gpssh`:

```
$ gpssh -h sdw1 -h sdw2 -v -e < command_file
```

Execute single commands in non-interactive mode on hosts `sdw2` and `localhost`:

```
$ gpssh -h sdw2 -h localhost -v -e 'ls -a /data/primary/*'
$ gpssh -h sdw2 -h localhost -v -e 'echo $GPHOME'
$ gpssh -h sdw2 -h localhost -v -e 'ls -1 | wc -l'
```

See Also

`gpssh-exkeys`, *`gpscp`*

gpssh-exkeys

Exchanges SSH public keys between hosts.

Synopsis

```
gpssh-exkeys -f hostfile_exkeys | -h hostname [-h hostname ...]

gpssh-exkeys -e hostfile_exkeys -x hostfile_gpexpand

gpssh-exkeys -?

gpssh-exkeys --version
```

Description

The `gpssh-exkeys` utility exchanges SSH keys between the specified host names (or host addresses). This allows SSH connections between Greenplum hosts and network interfaces without a password prompt. The utility is used to initially prepare a Greenplum Database system for passwordless SSH access, and also to prepare additional hosts for passwordless SSH access when expanding a Greenplum Database system.

Keys are exchanged as the currently logged in user. You run the utility on the master host as the `gpadmin` user (the user designated to own your Greenplum Database installation). Greenplum Database management utilities require that the `gpadmin` user be created on all hosts in the Greenplum Database system, and the utilities must be able to connect as that user to all hosts without a password prompt.

You can also use `gpssh-exkeys` to enable passwordless SSH for additional users, `root`, for example.

The `gpssh-exkeys` utility has the following prerequisites:

- The user must have an account on the master, standby, and every segment host in the Greenplum Database cluster.
- The user must have an `id_rsa` SSH key pair installed on the master host.
- The user must be able to connect with SSH from the master host to every other host machine without entering a password. (This is called "1-*n* passwordless SSH.")

You can enable 1-*n* passwordless SSH using the `ssh-copy-id` command to add the user's public key to each host's `authorized_keys` file. The `gpssh-exkeys` utility enables "*n*-*n* passwordless SSH," which allows the user to connect with SSH from any host to any other host in the cluster without a password.

To specify the hosts involved in an SSH key exchange, use the `-f` option to specify a file containing a list of host names (recommended), or use the `-h` option to name single host names on the command-line. At least one host name (`-h`) or a host file (`-f`) is required. Note that the local host is included in the key exchange by default.

To specify new expansion hosts to be added to an existing Greenplum Database system, use the `-e` and `-x` options. The `-e` option specifies a file containing a list of existing hosts in the system that have aready exchanged SSH keys. The `-x` option specifies a file containing a list of new hosts that need to participate in the SSH key exchange.

The `gpssh-exkeys` utility performs key exchange using the following steps:

- Adds the user's public key to the user's own `authorized_keys` file on the current host.
- Updates the `known_hosts` file of the current user with the host key of each host specified using the `-h`, `-f`, `-e`, and `-x` options.
- Connects to each host using `ssh` and obtains the user's `authorized_keys`, `known_hosts`, and `id_rsa.pub` files.

- Adds keys from the `id_rsa.pub` files obtained from each host to the `authorized_keys` file of the current user.
- Updates the `authorized_keys`, `known_hosts`, and `id_rsa.pub` files on all hosts with new host information (if any).

Options

`-e` *`hostfile_exkeys`*

When doing a system expansion, this is the name and location of a file containing all configured host names and host addresses (interface names) for each host in your *current* Greenplum system (master, standby master, and segments), one name per line without blank lines or extra spaces. Hosts specified in this file cannot be specified in the host file used with `-x`.

`-f` *`hostfile_exkeys`*

Specifies the name and location of a file containing all configured host names and host addresses (interface names) for each host in your Greenplum system (master, standby master and segments), one name per line without blank lines or extra spaces.

`-h` *`hostname`*

Specifies a single host name (or host address) that will participate in the SSH key exchange. You can use the `-h` option multiple times to specify multiple host names and host addresses.

`--version`

Displays the version of this utility.

`-x` *`hostfile_gpexpand`*

When doing a system expansion, this is the name and location of a file containing all configured host names and host addresses (interface names) for each *new segment host* you are adding to your Greenplum system, one name per line without blank lines or extra spaces. Hosts specified in this file cannot be specified in the host file used with `-e`.

`-? (help)`

Displays the online help.

Examples

Exchange SSH keys between all host names and addresses listed in the file `hostfile_exkeys`:

```
$ gpssh-exkeys -f hostfile_exkeys
```

Exchange SSH keys between the hosts `sdw1`, `sdw2`, and `sdw3`:

```
$ gpssh-exkeys -h sdw1 -h sdw2 -h sdw3
```

Exchange SSH keys between existing hosts `sdw1`, `sdw2`, and `sdw3`, and new hosts `sdw4` and `sdw5` as part of a system expansion operation:

```
$ cat hostfile_exkeys
mdw
mdw-1
mdw-2
smdw
smdw-1
smdw-2
sdw1
sdw1-1
sdw1-2
sdw2
```

```
sdw2-1
sdw2-2
sdw3
sdw3-1
sdw3-2
$ cat hostfile_gpexpand
sdw4
sdw4-1
sdw4-2
sdw5
sdw5-1
sdw5-2
$ gpssh-exkeys -e hostfile_exkeys -x hostfile_gpexpand
```

See Also

gpssh, *gpscp*

gpstart

Starts a Greenplum Database system.

Synopsis

```
gpstart [-d master_data_directory] [-B parallel_processes] [-R]
        [-m] [-y] [-a] [-t timeout_seconds] [-l logfile_directory]
        [--skip-heap-checksum-validation]
        [-v | -q]

gpstart -? | -h | --help

gpstart --version
```

Description

The `gpstart` utility is used to start the Greenplum Database server processes. When you start a Greenplum Database system, you are actually starting several `postgres` database server listener processes at once (the master and all of the segment instances). The `gpstart` utility handles the startup of the individual instances. Each instance is started in parallel.

The first time an administrator runs `gpstart`, the utility creates a hosts cache file named `.gphostcache` in the user's home directory. Subsequently, the utility uses this list of hosts to start the system more efficiently. If new hosts are added to the system, you must manually remove this file from the `gpadmin` user's home directory. The utility will create a new hosts cache file at the next startup.

As part of the startup process, the utility checks the consistency of heap checksum setting among the Greenplum Database master and segment instances, either enabled or disabled on all instances. If the heap checksum setting is different among the instances, an error is returned and Greenplum Database does not start. The validation can be disabled by specifying the option `--skip-heap-checksum-validation`. For more information about heap checksums, see *Enabling High Availability and Data Consistency Features* in the *Greenplum Database Admininstartor Guide*.

> **Note:** Before you can start a Greenplum Database system, you must have initialized the system using `gpinitsystem`. Enabling or disabing heap checksums is set when you initialize the system and cannot be changed after initialization.

If the Greenplum Database system is configured with a standby master, and `gpstart` does not detect it during startup, `gpstart` displays a warning and lets you cancel the startup operation.

- If the `-a` option (disable interactive mode prompts) is not specified, `gpstart` displays and logs these messages:

  ```
  Standby host is unreachable, cannot determine whether the standby is
   currently acting as the master. Received error: <error>
  Continue only if you are certain that the standby is not acting as the
   master.
  ```

  It also displays this prompt to continue startup:

  ```
  Continue with startup Yy|Nn (default=N):
  ```

- If the `-a` option is specified, the utility does not start the system. The messages are only logged, and `gpstart` adds this log message:

  ```
  Non interactive mode detected. Not starting the cluster. Start the cluster
   in interactive mode.
  ```

If the standby master is not accessible, you can start the system and troubleshoot standby master issues while the system is available.

Options

-a

Do not prompt the user for confirmation. Disables interactive mode.

-B *parallel_processes*

The number of segments to start in parallel. If not specified, the utility will start up to 64 parallel processes depending on how many segment instances it needs to start.

-d *master_data_directory*

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-m

Optional. Starts the master instance only, which may be useful for maintenance tasks. This mode only allows connections to the master in utility mode. For example:

```
PGOPTIONS='-c gp_session_role=utility' psql
```

The consistency of the heap checksum setting on master and segment instances is not checked.

-q

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-R

Starts Greenplum Database in restricted mode (only database superusers are allowed to connect).

--skip-heap-checksum-validation

During startup, the utility does not validate the consistency of the heap checksum setting among the Greenplum Database master and segment instances. The default is to ensure that the heap checksum setting is the same on all instances, either enabled or disabled.

> **Warning:** Starting Greenplum Database without this validation could lead to data loss. Use this option to start Greenplum Database only when it is

necessary to ignore the heap checksum verification errors to recover data or to troubleshoot the errors.

-t *timeout_seconds*

Specifies a timeout in seconds to wait for a segment instance to start up. If a segment instance was shutdown abnormally (due to power failure or killing its `postgres` database listener process, for example), it may take longer to start up due to the database recovery and validation process. If not specified, the default timeout is 60 seconds.

-v

Displays detailed status, progress and error messages output by the utility.

-y

Optional. Do not start the standby master host. The default is to start the standby master host and synchronization process.

-? | -h | --help

Displays the online help.

--version

Displays the version of this utility.

Examples

Start a Greenplum Database system:

```
gpstart
```

Start a Greenplum Database system in restricted mode (only allow superuser connections):

```
gpstart -R
```

Start the Greenplum master instance only and connect in utility mode:

```
gpstart -m PGOPTIONS='-c gp_session_role=utility' psql
```

See Also

gpstop, *gpinitsystem*

gpstate

Shows the status of a running Greenplum Database system.

Synopsis

```
gpstate [-d master_data_directory] [-B parallel_processes]
          [-s | -b | -Q | -e] [-m | -c] [-p] [-i] [-f] [-v | -q] | -x
          [-l log_directory]

gpstate -? | -h | --help
```

Description

The `gpstate` utility displays information about a running Greenplum Database instance. There is additional information you may want to know about a Greenplum Database system, since it is comprised of multiple PostgreSQL database instances (segments) spanning multiple machines. The `gpstate` utility provides additional status information for a Greenplum Database system, such as:

- Which segments are down.
- Master and segment configuration information (hosts, data directories, etc.).
- The ports used by the system.
- A mapping of primary segments to their corresponding mirror segments.

Options

-b (brief status)

Optional. Display a brief summary of the state of the Greenplum Database system. This is the default option.

-B *parallel_processes*

The number of segments to check in parallel. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances it needs to check.

-c (show primary to mirror mappings)

Optional. Display mapping of primary segments to their corresponding mirror segments.

-d *master_data_directory*

Optional. The master data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-e (show segments with mirror status issues)

Show details on primary/mirror segment pairs that have potential issues such as 1) the active segment is running in change tracking mode, meaning a segment is down 2) the active segment is in resynchronization mode, meaning it is catching up changes to the mirror 3) a segment is not in its preferred role, for example a segment that was a primary at system initialization time is now acting as a mirror, meaning you may have one or more segment hosts with unbalanced processing load.

-f (show standby master details)

Display details of the standby master host if configured.

-i (show Greenplum Database version)

Display the Greenplum Database software version information for each instance.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-m (list mirrors)

Optional. List the mirror segment instances in the system, their current role, and synchronization status.

-p (show ports)

List the port numbers used throughout the Greenplum Database system.

-q (no screen output)

Optional. Run in quiet mode. Except for warning messages, command output is not displayed on the screen. However, this information is still written to the log file.

-Q (quick status)

Optional. Checks segment status in the system catalog on the master host. Does not poll the segments for status.

-s (detailed status)

Optional. Displays detailed status information for the Greenplum Database system.

-v (verbose output)

Optional. Displays error messages and outputs detailed status and progress information.

-x (expand)

Optional. Displays detailed information about the progress and state of a Greenplum system expansion.

-? | -h | --help (help)

Displays the online help.

Output Field Definitions

The following output fields are reported by `gpstate -s` for the master:

Table 66: gpstate output data for the master

Output Data	Description
Master host	host name of the master
Master postgres process ID	PID of the master database listener process
Master data directory	file system location of the master data directory
Master port	port of the master `postgres` database listener process
Master current role	dispatch = regular operating mode utility = maintenance mode
Greenplum array configuration type	Standard = one NIC per host Multi-Home = multiple NICs per host
Greenplum initsystem version	version of Greenplum Database when system was first initialized
Greenplum current version	current version of Greenplum Database
Postgres version	version of PostgreSQL that Greenplum Database is based on
Greenplum mirroring status	physical mirroring or none
Master standby	host name of the standby master
Standby master state	status of the standby master: active or passive

The following output fields are reported by `gpstate -s` for each segment:

Table 67: gpstate output data for segments

Output Data	Description
Hostname	system-configured host name
Address	network address host name (NIC name)
Datadir	file system location of segment data directory
Port	port number of segment `postgres` database listener process
Current Role	current role of a segment: *Mirror* or *Primary*
Preferred Role	role at system initialization time: *Mirror* or *Primary*

Output Data	Description
Mirror Status	status of a primary/mirror segment pair: *Synchronized* = data is up to date on both *Resynchronization* = data is currently being copied from one to the other *Change Tracking* = segment down and active segment is logging changes
Change tracking data size	when in *Change Tracking* mode, the size of the change log file (may grow and shrink as compression is applied)
Estimated total data to synchronize	when in *Resynchronization* mode, the estimated size of data left to synchronize
Data synchronized	when in *Resynchronization* mode, the estimated size of data that has already been synchronized
Estimated resync progress with mirror	When in *Resynchronization* mode, the estimated percentage of completion
Estimated resync end time	when in *Resynchronization* mode, the estimated time to complete
File `postmaster.pid`	status of `postmaster.pid` lock file: *Found* or *Missing*
PID from `postmaster.pid` file	PID found in the `postmaster.pid` file
Lock files in `/tmp`	a segment port lock file for its `postgres` process is created in `/tmp` (file is removed when a segment shuts down)
Active PID	active process ID of a segment
Master reports status as	segment status as reported in the system catalog: *Up* or *Down*
Database status	status of Greenplum Database to incoming requests: *Up*, *Down*, or *Suspended*. A *Suspended* state means database activity is temporarily paused while a segment transitions from one state to another.

The following output fields are reported by `gpstate -f` for standby master replication status:

Table 68: gpstate output data for master replication

Output Data	Description
Standby address	hostname of the standby master
Standby data dir	file system location of the standby master data directory
Standby port	port of the standby master `postgres` database listener process
Standby PID	process ID of the standby master

Output Data	Description
Standby status	status of the standby master: *Standby host passive*
WAL Sender State	write-ahead log (WAL) streaming state: *streaming*, *startup*,*backup*, *catchup*
Sync state	WAL sender synchronization state: *sync*
Sent Location	WAL sender transaction log (xlog) record sent location
Flush Location	WAL receiver xlog record flush location
Replay Location	standby xlog record replay location

Examples

Show detailed status information of a Greenplum Database system:

```
gpstate -s
```

Do a quick check for down segments in the master host system catalog:

```
gpstate -Q
```

Show information about mirror segment instances:

```
gpstate -m
```

Show information about the standby master configuration:

```
gpstate -f
```

Display the Greenplum software version information:

```
gpstate -i
```

See Also

gpstart, *gpexpandgplogfilter*

gpstop

Stops or restarts a Greenplum Database system.

Synopsis

```
gpstop [-d master_data_directory] [-B parallel_processes]
       [-M smart | fast | immediate] [-t timeout_seconds] [-r] [-y] [-a]
       [-l logfile_directory] [-v | -q]

gpstop -m [-d master_data_directory] [-y] [-l logfile_directory] [-v | -q]

gpstop -u [-d master_data_directory] [-l logfile_directory] [-v | -q]

gpstop --host host_name [-d master_data_directory] [-l logfile_directory]
       [-t timeout_seconds] [-a] [-v | -q]

gpstop --version
```

```
gpstop -? | -h | --help
```

Description

The `gpstop` utility is used to stop the database servers that comprise a Greenplum Database system. When you stop a Greenplum Database system, you are actually stopping several `postgres` database server processes at once (the master and all of the segment instances). The `gpstop` utility handles the shutdown of the individual instances. Each instance is shutdown in parallel.

The default shutdown mode (`-M smart`) waits for current client connections to finish before completing the shutdown. If any connections remain open after the timeout period, or if you interrupt with CTRL-C, `gpstop` lists the open connections and prompts whether to continue waiting for connections to finish, or to perform a fast or immediate shutdown. The default timeout period is 120 seconds and can be changed with the `-t` *`timeout_seconds`* option.

Specify the `-M fast` shutdown mode to roll back all in-progress transactions and terminate any connections before shutting down.

With the `-u` option, the utility uploads changes made to the master `pg_hba.conf` file or to *runtime* configuration parameters in the master `postgresql.conf` file without interruption of service. Note that any active sessions will not pickup the changes until they reconnect to the database.

Options

`-a`

Do not prompt the user for confirmation.

`-B` *`parallel_processes`*

The number of segments to stop in parallel. If not specified, the utility will start up to 64 parallel processes depending on how many segment instances it needs to stop.

`-d` *`master_data_directory`*

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

`--host` *`host_name`*

The utility shuts down the Greenplum Database segment instances on the specified host to allow maintenance on the host. Each primary segment instance on the host is shut down and the associated mirror segment instance is promoted to a primary segment if the mirror segment is on another host. Mirror segment instances on the host are shut down.

The segment instances are not shut down and the utility returns an error in these cases:

- Segment mirroring is not enabled for the system.
- The master or standby master is on the host.
- Both a primary segment instance and its mirror are on the host.

This option cannot be specified with the `-m`, `-r`, `-u`, or `-y` options.

> **Note:** The `gprecoverseg` utility restores segment instances. Run `gprecoverseg` commands to start the segments as mirrors and then to return the segments to their preferred role (primary segments).

`-l` *`logfile_directory`*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

`-m`

Optional. Shuts down a Greenplum master instance that was started in maintenance mode.

`-M fast`

Fast shut down. Any transactions in progress are interrupted and rolled back.

-M immediate

Immediate shut down. Any transactions in progress are aborted.

This mode kills all `postgres` processes without allowing the database server to complete transaction processing or clean up any temporary or in-process work files.

-M smart

Smart shut down. This is the default shutdown mode. `gpstop` waits for active user connections to disconnect and then proceeds with the shutdown. If any user connections remain open after the timeout period (or if you interrupt by pressing CTRL-C) `gpstop` lists the open user connections and prompts whether to continue waiting for connections to finish, or to perform a fast or immediate shutdown.

-q

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-r

Restart after shutdown is complete.

-t *timeout_seconds*

Specifies a timeout threshold (in seconds) to wait for a segment instance to shutdown. If a segment instance does not shutdown in the specified number of seconds, `gpstop` displays a message indicating that one or more segments are still in the process of shutting down and that you cannot restart Greenplum Database until the segment instance(s) are stopped. This option is useful in situations where `gpstop` is executed and there are very large transactions that need to rollback. These large transactions can take over a minute to rollback and surpass the default timeout period of 120 seconds.

-u

This option reloads the `pg_hba.conf` files of the master and segments and the runtime parameters of the `postgresql.conf` files but does not shutdown the Greenplum Database array. Use this option to make new configuration settings active after editing `postgresql.conf` or `pg_hba.conf`. Note that this only applies to configuration parameters that are designated as *runtime* parameters.

-v

Displays detailed status, progress and error messages output by the utility.

-y

Do not stop the standby master process. The default is to stop the standby master.

-? | -h | --help

Displays the online help.

--version

Displays the version of this utility.

Examples

Stop a Greenplum Database system in smart mode:

```
gpstop
```

Stop a Greenplum Database system in fast mode:

```
gpstop -M fast
```

Stop all segment instances and then restart the system:

```
gpstop -r
```

Stop a master instance that was started in maintenance mode:

```
gpstop -m
```

Reload the `postgresql.conf` and `pg_hba.conf` files after making configuration changes but do not shutdown the Greenplum Database array:

```
gpstop -u
```

See Also

gpstart

pg_config

Retrieves information about the installed version of Greenplum Database.

Synopsis

```
pg_config [option ...]

pg_config -? | --help

pg_config --version
```

Description

The `pg_config` utility prints configuration parameters of the currently installed version of Greenplum Database. It is intended, for example, to be used by software packages that want to interface to Greenplum Database to facilitate finding the required header files and libraries. Note that information printed out by `pg_config` is for the Greenplum Database master only.

If more than one option is given, the information is printed in that order, one item per line. If no options are given, all available information is printed, with labels.

Options

`--bindir`

Print the location of user executables. Use this, for example, to find the `psql` program. This is normally also the location where the `pg_config` program resides.

`--docdir`

Print the location of documentation files.

`--includedir`

Print the location of C header files of the client interfaces.

`--pkgincludedir`

Print the location of other C header files.

`--includedir-server`

Print the location of C header files for server programming.

`--libdir`

Print the location of object code libraries.

--pkglibdir

Print the location of dynamically loadable modules, or where the server would search for them. (Other architecture-dependent data files may also be installed in this directory.)

--localedir

Print the location of locale support files.

--mandir

Print the location of manual pages.

--sharedir

Print the location of architecture-independent support files.

--sysconfdir

Print the location of system-wide configuration files.

--pgxs

Print the location of extension makefiles.

--configure

Print the options that were given to the configure script when Greenplum Database was configured for building.

--cc

Print the value of the CC variable that was used for building Greenplum Database. This shows the C compiler used.

--cppflags

Print the value of the `CPPFLAGS` variable that was used for building Greenplum Database. This shows C compiler switches needed at preprocessing time.

--cflags

Print the value of the `CFLAGS` variable that was used for building Greenplum Database. This shows C compiler switches.

--cflags_sl

Print the value of the `CFLAGS_SL` variable that was used for building Greenplum Database. This shows extra C compiler switches used for building shared libraries.

--ldflags

Print the value of the `LDFLAGS` variable that was used for building Greenplum Database. This shows linker switches.

--ldflags_ex

Print the value of the `LDFLAGS_EX` variable that was used for building Greenplum Database. This shows linker switches that were used for building executables only.

--ldflags_sl

Print the value of the `LDFLAGS_SL` variable that was used for building Greenplum Database. This shows linker switches used for building shared libraries only.

--libs

Print the value of the `LIBS` variable that was used for building Greenplum Database. This normally contains `-l` switches for external libraries linked into Greenplum Database.

--version

Print the version of Greenplum Database.

Examples

To reproduce the build configuration of the current Greenplum Database installation, run the following command:

```
eval ./configure 'pg_config --configure'
```

The output of `pg_config --configure` contains shell quotation marks so arguments with spaces are represented correctly. Therefore, using `eval` is required for proper results.

pg_dump

Extracts a database into a single script file or other archive file.

Synopsis

```
pg_dump [connection-option ...] [dump_option ...] [dbname]

pg_dump -? | --help

pg_dump -V | --version
```

Description

`pg_dump` is a standard PostgreSQL utility for backing up a database, and is also supported in Greenplum Database. It creates a single (non-parallel) dump file. For routine backups of Greenplum Database, it is better to use the Greenplum Database backup utility, *`gpbackup`*, for the best performance.

Use `pg_dump` if you are migrating your data to another database vendor's system, or to another Greenplum Database system with a different segment configuration (for example, if the system you are migrating to has greater or fewer segment instances). To restore, you must use the corresponding *`pg_restore`* utility (if the dump file is in archive format), or you can use a client program such as *`psql`* (if the dump file is in plain text format).

Since `pg_dump` is compatible with regular PostgreSQL, it can be used to migrate data into Greenplum Database. The `pg_dump` utility in Greenplum Database is very similar to the PostgreSQL `pg_dump` utility, with the following exceptions and limitations:

- If using `pg_dump` to backup a Greenplum Database database, keep in mind that the dump operation can take a long time (several hours) for very large databases. Also, you must make sure you have sufficient disk space to create the dump file.
- If you are migrating data from one Greenplum Database system to another, use the `--gp-syntax` command-line option to include the `DISTRIBUTED BY` clause in `CREATE TABLE` statements. This ensures that Greenplum Database table data is distributed with the correct distribution key columns upon restore.

`pg_dump` makes consistent backups even if the database is being used concurrently. `pg_dump` does not block other users accessing the database (readers or writers).

When used with one of the archive file formats and combined with `pg_restore`, `pg_dump` provides a flexible archival and transfer mechanism. `pg_dump` can be used to backup an entire database, then `pg_restore` can be used to examine the archive and/or select which parts of the database are to be restored. The most flexible output file formats are the *custom* format (`-Fc`) and the *directory* format (`-Fd`). They allow for selection and reordering of all archived items, support parallel restoration, and are compressed by default. The *directory* format is the only format that supports parallel dumps.

Options

`dbname`

Specifies the name of the database to be dumped. If this is not specified, the environment variable PGDATABASE is used. If that is not set, the user name specified for the connection is used.

Dump Options

-a | --data-only

Dump only the data, not the schema (data definitions). Table data and sequence values are dumped.

This option is similar to, but for historical reasons not identical to, specifying `--section=data`.

-b | --blobs

Include large objects in the dump. This is the default behavior except when `--schema`, `--table`, or `--schema-only` is specified. The `-b` switch is only useful add large objects to dumps where a specific schema or table has been requested. Note that blobs are considered data and therefore will be included when `--data-only` is used, but not when `--schema-only` is.

> **Note:** Greenplum Database does not support the PostgreSQL *large object facility* for streaming user data that is stored in large-object structures.

-c | --clean

Adds commands to the text output file to clean (drop) database objects prior to outputting the commands for creating them. (Restore might generate some harmless error messages, if any objects were not present in the destination database.) Note that objects are not dropped before the dump operation begins, but DROP commands are added to the DDL dump output files so that when you use those files to do a restore, the DROP commands are run prior to the CREATE commands. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call *pg_restore*.

-C | --create

Begin the output with a command to create the database itself and reconnect to the created database. (With a script of this form, it doesn't matter which database in the destination installation you connect to before running the script.) If `--clean` is also specified, the script drops and recreates the target database before reconnecting to it. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call *pg_restore*.

-E *encoding* | --encoding=*encoding*

Create the dump in the specified character set encoding. By default, the dump is created in the database encoding. (Another way to get the same result is to set the PGCLIENTENCODING environment variable to the desired dump encoding.)

-f *file* | --file=*file*

Send output to the specified file. This parameter can be omitted for file-based output formats, in which case the standard output is used. It must be given for the directory output format however, where it specifies the target directory instead of a file. In this case the directory is created by pg_dump and must not exist before.

-F p|c|d|t | --format=plain|custom|directory|tar

Selects the format of the output. format can be one of the following:

p | plain — Output a plain-text SQL script file (the default).

c | custom — Output a custom archive suitable for input into *pg_restore*. Together with the directory output format, this is the most flexible output format in that it allows manual selection and reordering of archived items during restore. This format is compressed by default and also supports parallel dumps.

d | directory — Output a directory-format archive suitable for input into `pg_restore`. This will create a directory with one file for each table and blob being dumped, plus a so-called Table of Contents file describing the dumped objects in a machine-readable format that `pg_restore` can read. A directory format archive can be manipulated with standard Unix tools; for example, files in an uncompressed archive can be compressed with the `gzip` tool. This format is compressed by default.

t | tar — Output a tar-format archive suitable for input into *`pg_restore`*. The tar format is compatible with the directory format; extracting a tar-format archive produces a valid directory-format archive. However, the tar format does not support compression. Also, when using tar format the relative order of table data items cannot be changed during restore.

`-j` *`njobs`* | `--jobs=`*`njobs`*

Run the dump in parallel by dumping *njobs* tables simultaneously. This option reduces the time of the dump but it also increases the load on the database server. You can only use this option with the directory output format because this is the only output format where multiple processes can write their data at the same time.

> **Note:** Parallel dumps using `pg_dump` are parallelized only on the query dispatcher (master) node, not across the query executor (segment) nodes as is the case when you use `gpbackup`.

`pg_dump` will open *njobs* + 1 connections to the database, so make sure your *`max_connections`* setting is high enough to accommodate all connections.

Requesting exclusive locks on database objects while running a parallel dump could cause the dump to fail. The reason is that the `pg_dump` master process requests shared locks on the objects that the worker processes are going to dump later in order to make sure that nobody deletes them and makes them go away while the dump is running. If another client then requests an exclusive lock on a table, that lock will not be granted but will be queued waiting for the shared lock of the master process to be released. Consequently, any other access to the table will not be granted either and will queue after the exclusive lock request. This includes the worker process trying to dump the table. Without any precautions this would be a classic deadlock situation. To detect this conflict, the `pg_dump` worker process requests another shared lock using the `NOWAIT` option. If the worker process is not granted this shared lock, somebody else must have requested an exclusive lock in the meantime and there is no way to continue with the dump, so `pg_dump` has no choice but to abort the dump.

For a consistent backup, the database server needs to support synchronized snapshots, a feature that was introduced in Greenplum Database 6.0. With this feature, database clients can ensure they see the same data set even though they use different connections. `pg_dump -j` uses multiple database connections; it connects to the database once with the master process and once again for each worker job. Without the synchronized snapshot feature, the different worker jobs wouldn't be guaranteed to see the same data in each connection, which could lead to an inconsistent backup.

If you want to run a parallel dump of a pre-6.0 server, you need to make sure that the database content doesn't change from between the time the master connects to the database until the last worker job has connected to the database. The easiest way to do this is to halt any data modifying processes (DDL and DML) accessing the database before starting the backup. You also need to specify the `--no-synchronized-snapshots` parameter when running `pg_dump -j` against a pre-6.0 Greenplum Database server.

`-n` *`schema`* | `--schema=`*`schema`*

Dump only schemas matching the schema pattern; this selects both the schema itself, and all its contained objects. When this option is not specified, all non-system schemas in the target database will be dumped. Multiple schemas can be selected by writing multiple `-n` switches. Also, the schema parameter is interpreted as a pattern according to the same

rules used by `psql`'s `\d` commands, so multiple schemas can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards.

Note: When -n is specified, `pg_dump` makes no attempt to dump any other database objects that the selected schema(s) may depend upon. Therefore, there is no guarantee that the results of a specific-schema dump can be successfully restored by themselves into a clean database.

> **Note:** Non-schema objects such as blobs are not dumped when `-n` is specified. You can add blobs back to the dump with the `--blobs` switch.

`-N` *`schema`* | `--exclude-schema=`*`schema`*

Do not dump any schemas matching the schema pattern. The pattern is interpreted according to the same rules as for `-n`. `-N` can be given more than once to exclude schemas matching any of several patterns. When both `-n` and `-N` are given, the behavior is to dump just the schemas that match at least one `-n` switch but no `-N` switches. If `-N` appears without `-n`, then schemas matching `-N` are excluded from what is otherwise a normal dump.

`-o` | `--oids`

Dump object identifiers (OIDs) as part of the data for every table. Use of this option is not recommended for files that are intended to be restored into Greenplum Database.

`-O` | `--no-owner`

Do not output commands to set ownership of objects to match the original database. By default, `pg_dump` issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created database objects. These statements will fail when the script is run unless it is started by a superuser (or the same user that owns all of the objects in the script). To make a script that can be restored by any user, but will give that user ownership of all the objects, specify `-O`. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call *`pg_restore`*.

`-s` | `--schema-only`

Dump only the object definitions (schema), not data.

This option is the inverse of `--data-only`. It is similar to, but for historical reasons not identical to, specifying `--section=pre-data --section=post-data`.

(Do not confuse this with the `--schema` option, which uses the word "schema" in a different meaning.)

To exclude table data for only a subset of tables in the database, see `--exclude-table-data`.

`-S` *`username`* | `--superuser=`*`username`*

Specify the superuser user name to use when disabling triggers. This is relevant only if `--disable-triggers` is used. It is better to leave this out, and instead start the resulting script as a superuser.

> **Note:** Greenplum Database does not support user-defined triggers.

`-t` *`table`* | `--table=`*`table`*

Dump only tables (or views or sequences or foreign tables) matching the table pattern. Specify the table in the format `schema.table`.

Multiple tables can be selected by writing multiple `-t` switches. Also, the table parameter is interpreted as a pattern according to the same rules used by `psql`'s `\d` commands, so multiple tables can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards. The `-n` and `-N` switches have no effect when `-t` is used,

because tables selected by `-t` will be dumped regardless of those switches, and non-table objects will not be dumped.

> **Note:** When `-t` is specified, `pg_dump` makes no attempt to dump any other database objects that the selected table(s) may depend upon. Therefore, there is no guarantee that the results of a specific-table dump can be successfully restored by themselves into a clean database.
>
> Also, `-t` cannot be used to specify a child table partition. To dump a partitioned table, you must specify the parent table name.

`-T` *`table`* `| --exclude-table=`*`table`*

Do not dump any tables matching the table pattern. The pattern is interpreted according to the same rules as for `-t`. `-T` can be given more than once to exclude tables matching any of several patterns. When both `-t` and `-T` are given, the behavior is to dump just the tables that match at least one `-t` switch but no `-T` switches. If `-T` appears without `-t`, then tables matching `-T` are excluded from what is otherwise a normal dump.

`-v | --verbose`

Specifies verbose mode. This will cause `pg_dump` to output detailed object comments and start/stop times to the dump file, and progress messages to standard error.

`-V | --version`

Print the `pg_dump` version and exit.

`-x | --no-privileges | --no-acl`

Prevent dumping of access privileges (`GRANT/REVOKE` commands).

`-Z 0..9 | --compress=0..9`

Specify the compression level to use. Zero means no compression. For the custom archive format, this specifies compression of individual table-data segments, and the default is to compress at a moderate level.

For plain text output, setting a non-zero compression level causes the entire output file to be compressed, as though it had been fed through `gzip`; but the default is not to compress. The tar archive format currently does not support compression at all.

`--binary-upgrade`

This option is for use by in-place upgrade utilities. Its use for other purposes is not recommended or supported. The behavior of the option may change in future releases without notice.

`--column-inserts | --attribute-inserts`

Dump data as `INSERT` commands with explicit column names (`INSERT INTO`*`table`*`(`*`column`*`, ...) VALUES ...`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL-based databases. However, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents.

`--disable-dollar-quoting`

This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

`--disable-triggers`

This option is relevant only when creating a data-only dump. It instructs `pg_dump` to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have triggers on the tables that you do not want to invoke during data reload. The commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably be careful

to start the resulting script as a superuser. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call *pg_restore*.

Note: Greenplum Database does not support user-defined triggers.

--exclude-table-data=*table*

Do not dump data for any tables matching the *table* pattern. The pattern is interpreted according to the same rules as for `-t`. `--exclude-table-data` can be given more than once to exclude tables matching any of several patterns. This option is useful when you need the definition of a particular table even though you do not need the data in it.

To exclude data for all tables in the database, see `--schema-only`.

--if-exists

Use conditional commands (i.e. add an `IF EXISTS` clause) when cleaning database objects. This option is not valid unless `--clean` is also specified.

--inserts

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL-based databases. However, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents. Note that the restore may fail altogether if you have rearranged column order. The `--column-inserts` option is safe against column order changes, though even slower.

--lock-wait-timeout=*timeout*

Do not wait forever to acquire shared table locks at the beginning of the dump. Instead, fail if unable to lock a table within the specified *timeout*. Specify *timeout* as a number of milliseconds.

--no-security-labels

Do not dump security labels.

--no-synchronized-snapshots

This option allows running `pg_dump -j` against a pre-6.0 Greenplum Database server; see the documentation of the `-j` parameter for more details.

--no-tablespaces

Do not output commands to select tablespaces. With this option, all objects will be created in whichever tablespace is the default during restore.

This option is only meaningful for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

--no-unlogged-table-data

Do not dump the contents of unlogged tables. This option has no effect on whether or not the table definitions (schema) are dumped; it only suppresses dumping the table data. Data in unlogged tables is always excluded when dumping from a standby server.

--quote-all-identifiers

Force quoting of all identifiers. This option is recommended when dumping a database from a server whose Greenplum Database major version is different from `pg_dump`'s, or when the output is intended to be loaded into a server of a different major version. By default, `pg_dump` quotes only identifiers that are reserved words in its own major version. This sometimes results in compatibility issues when dealing with servers of other versions that may have slightly different sets of reserved words. Using `--quote-all-identifiers` prevents such issues, at the price of a harder-to-read dump script.

--section=*sectionname*

Only dump the named section. The section name can be `pre-data`, `data`, or `post-data`. This option can be specified more than once to select multiple sections. The default is to dump all sections.

The `data` section contains actual table data and sequence values. `post-data` items include definitions of indexes, triggers, rules, and constraints other than validated check constraints. `pre-data` items include all other data definition items.

`--serializable-deferrable`

Use a serializable transaction for the dump, to ensure that the snapshot used is consistent with later database states; but do this by waiting for a point in the transaction stream at which no anomalies can be present, so that there isn't a risk of the dump failing or causing other transactions to roll back with a serialization_failure.

This option is not beneficial for a dump which is intended only for disaster recovery. It could be useful for a dump used to load a copy of the database for reporting or other read-only load sharing while the original database continues to be updated. Without it the dump may reflect a state which is not consistent with any serial execution of the transactions eventually committed. For example, if batch processing techniques are used, a batch may show as closed in the dump without all of the items which are in the batch appearing.

This option will make no difference if there are no read-write transactions active when `pg_dump` is started. If read-write transactions are active, the start of the dump may be delayed for an indeterminate length of time. Once running, performance with or without the switch is the same.

> **Note:** Because Greenplum Database does not support serializable transactions, the `--serializable-deferrable` option has no effect in Greenplum Database.

`--use-set-session-authorization`

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards-compatible, but depending on the history of the objects in the dump, may not restore properly. A dump using `SET SESSION AUTHORIZATION` will require superuser privileges to restore correctly, whereas `ALTER OWNER` requires lesser privileges.

`--gp-syntax | --no-gp-syntax`

Use `--gp-syntax` to dump Greenplum Database syntax in the `CREATE TABLE` statements. This allows the distribution policy (`DISTRIBUTED BY` or `DISTRIBUTED RANDOMLY` clauses) of a Greenplum Database table to be dumped, which is useful for restoring into other Greenplum Database systems. The default is to include Greenplum Database syntax when connected to a Greenplum Database system, and to exclude it when connected to a regular PostgreSQL system.

`--function-oids` *`oids`*

Dump the function(s) specified in the *oids* list of object identifiers.

> **Note:** This option is provided solely for use by other administration utilities; its use for any other purpose is not recommended or supported. The behaviour of the option may change in future releases without notice.

`--relation-oids` *`oids`*

Dump the relation(s) specified in the *oids* list of object identifiers.

> **Note:** This option is provided solely for use by other administration utilities; its use for any other purpose is not recommended or supported. The behaviour of the option may change in future releases without notice.

`-? | --help`

Show help about `pg_dump` command line arguments, and exit.

Connection Options

-d *dbname* | --dbname=*dbname*

Specifies the name of the database to connect to. This is equivalent to specifying *dbname* as the first non-option argument on the command line.

If this parameter contains an = sign or starts with a valid URI prefix (`postgresql://` or `postgres://`), it is treated as a *conninfo* string. See *Connection Strings* in the PostgreSQL documentation for more information.

-h *host* | --host=*host*

The host name of the machine on which the Greenplum Database master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

-p *port* | --port=*port*

The TCP port on which the Greenplum Database master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username=*username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-W | --password

Force a password prompt.

-w | --no-password

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

--role=*rolename*

Specifies a role name to be used to create the dump. This option causes `pg_dump` to issue a `SET ROLE` *rolename* command after connecting to the database. It is useful when the authenticated user (specified by `-U`) lacks privileges needed by `pg_dump`, but can switch to a role with the required rights. Some installations have a policy against logging in directly as a superuser, and use of this option allows dumps to be made without violating the policy.

Notes

When a data-only dump is chosen and the option `--disable-triggers` is used, `pg_dump` emits commands to disable triggers on user tables before inserting the data and commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs may be left in the wrong state.

The dump file produced by `pg_dump` does not contain the statistics used by the optimizer to make query planning decisions. Therefore, it is wise to run `ANALYZE` after restoring from a dump file to ensure optimal performance.

The database activity of `pg_dump` is normally collected by the statistics collector. If this is undesirable, you can set parameter `track_counts` to false via `PGOPTIONS` or the `ALTER USER` command.

Because `pg_dump` may be used to transfer data to newer versions of Greenplum Database, the output of `pg_dump` can be expected to load into Greenplum Database versions newer than `pg_dump`'s version. `pg_dump` can also dump from Greenplum Database versions older than its own version. However, `pg_dump` cannot dump from Greenplum Database versions newer than its own major version; it will refuse to even try, rather than risk making an invalid dump. Also, it is not guaranteed that `pg_dump`'s output can be loaded into a server of an older major version — not even if the dump was taken from a server

of that version. Loading a dump file into an older server may require manual editing of the dump file to remove syntax not understood by the older server. Use of the `--quote-all-identifiers` option is recommended in cross-version cases, as it can prevent problems arising from varying reserved-word lists in different Greenplum Database versions.

Examples

Dump a database called `mydb` into a SQL-script file:

```
pg_dump mydb > db.sql
```

To reload such a script into a (freshly created) database named `newdb`:

```
psql -d newdb -f db.sql
```

Dump a Greenplum Database in tar file format and include distribution policy information:

```
pg_dump -Ft --gp-syntax mydb > db.tar
```

To dump a database into a custom-format archive file:

```
pg_dump -Fc mydb > db.dump
```

To dump a database into a directory-format archive:

```
pg_dump -Fd mydb -f dumpdir
```

To dump a database into a directory-format archive in parallel with 5 worker jobs:

```
pg_dump -Fd mydb -j 5 -f dumpdir
```

To reload an archive file into a (freshly created) database named `newdb`:

```
pg_restore -d newdb db.dump
```

To dump a single table named `mytab`:

```
pg_dump -t mytab mydb > db.sql
```

To specify an upper-case or mixed-case name in `-t` and related switches, you need to double-quote the name; else it will be folded to lower case. But double quotes are special to the shell, so in turn they must be quoted. Thus, to dump a single table with a mixed-case name, you need something like:

```
pg_dump -t '"MixedCaseName"' mydb > mytab.sql
```

See Also

`pg_dumpall`, *`pg_restore`*, *`psql`*

pg_dumpall

Extracts all databases in a Greenplum Database system to a single script file or other archive file.

Synopsis

```
pg_dumpall [connection-option ...] [dump_option ...]
```

```
pg_dumpall -? | --help

pg_dumpall -V | --version
```

Description

`pg_dumpall` is a standard PostgreSQL utility for backing up all databases in a Greenplum Database (or PostgreSQL) instance, and is also supported in Greenplum Database. It creates a single (non-parallel) dump file. For routine backups of Greenplum Database it is better to use the Greenplum Database backup utility, *gpbackup*, for the best performance.

`pg_dumpall` creates a single script file that contains SQL commands that can be used as input to *psql* to restore the databases. It does this by calling *pg_dump* for each database. `pg_dumpall` also dumps global objects that are common to all databases. (`pg_dump` does not save these objects.) This currently includes information about database users and groups, and access permissions that apply to databases as a whole.

Since `pg_dumpall` reads tables from all databases you will most likely have to connect as a database superuser in order to produce a complete dump. Also you will need superuser privileges to execute the saved script in order to be allowed to add users and groups, and to create databases.

The SQL script will be written to the standard output. Use the `[-f | --file]` option or shell operators to redirect it into a file.

`pg_dumpall` needs to connect several times to the Greenplum Database master server (once per database). If you use password authentication it is likely to ask for a password each time. It is convenient to have a `~/.pgpass` file in such cases.

Options

Dump Options

`-a | --data-only`

Dump only the data, not the schema (data definitions). This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call *pg_restore*.

`-c | --clean`

Output commands to clean (drop) database objects prior to (the commands for) creating them. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call *pg_restore*.

`-f` *`filename`* `| --file=`*`filename`

Send output to the specified file.

`-g | --globals-only`

Dump only global objects (roles and tablespaces), no databases.

`-o | --oids`

Dump object identifiers (OIDs) as part of the data for every table. Use of this option is not recommended for files that are intended to be restored into Greenplum Database.

`-O | --no-owner`

Do not output commands to set ownership of objects to match the original database. By default, *pg_dump* issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created database objects. These statements will fail when the script is run unless it is started by a superuser (or the same user that owns all of the objects in the script). To make a script that can be restored by any user, but will give that user ownership of all the objects, specify `-O`. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call *pg_restore*.

`-r | --roles-only`

Dump only roles, not databases or tablespaces.

-s | --schema-only

Dump only the object definitions (schema), not data.

-S *username* | --superuser=*username*

Specify the superuser user name to use when disabling triggers. This is relevant only if `--disable-triggers` is used. It is better to leave this out, and instead start the resulting script as a superuser.

Note: Greenplum Database does not support user-defined triggers.

-t | --tablespaces-only

Dump only tablespaces, not databases or roles.

-v | --verbose

Specifies verbose mode. This will cause *pg_dump* to output detailed object comments and start/stop times to the dump file, and progress messages to standard error.

-V | --version

Print the `pg_dumpall` version and exit.

-x | --no-privileges | --no-acl

Prevent dumping of access privileges (`GRANT/REVOKE` commands).

--binary-upgrade

This option is for use by in-place upgrade utilities. Its use for other purposes is not recommended or supported. The behavior of the option may change in future releases without notice.

--column-inserts | --attribute-inserts

Dump data as `INSERT` commands with explicit column names (`INSERT INTO` *`table`* `(`*`column`*`, ...) VALUES ...`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL-based databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents.

--disable-dollar-quoting

This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

--disable-triggers

This option is relevant only when creating a data-only dump. It instructs `pg_dumpall` to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have triggers on the tables that you do not want to invoke during data reload. The commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably be careful to start the resulting script as a superuser.

Note: Greenplum Database does not support user-defined triggers.

--inserts

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL-based databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents. Note that the restore may fail altogether if you have rearranged column order. The `--column-inserts` option is safe against column order changes, though even slower.

--lock-wait-timeout=*timeout*

Do not wait forever to acquire shared table locks at the beginning of the dump. Instead, fail if unable to lock a table within the specified *timeout*. The timeout may be specified in any

of the formats accepted by `SET statement_timeout`. Allowed values vary depending on the server version you are dumping from, but an integer number of milliseconds is accepted by all Greenplum Database versions.

`--no-security-labels`

Do not dump security labels.

`--no-tablespaces`

Do not output commands to select tablespaces. With this option, all objects will be created in whichever tablespace is the default during restore.

`--no-unlogged-table-data`

Do not dump the contents of unlogged tables. This option has no effect on whether or not the table definitions (schema) are dumped; it only suppresses dumping the table data.

`--quote-all-identifiers`

Force quoting of all identifiers. This option is recommended when dumping a database from a server whose Greenplum Database major version is different from `pg_dumpall`'s, or when the output is intended to be loaded into a server of a different major version. By default, `pg_dumpall` quotes only identifiers that are reserved words in its own major version. This sometimes results in compatibility issues when dealing with servers of other versions that may have slightly different sets of reserved words. Using `--quote-all-identifiers` prevents such issues, at the price of a harder-to-read dump script.

`--resource-queues`

Dump resource queue definitions.

`--resource-groups`

Dump resource group definitions.

`--use-set-session-authorization`

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, may not restore properly. A dump using `SET SESSION AUTHORIZATION` will require superuser privileges to restore correctly, whereas `ALTER OWNER` requires lesser privileges.

`--gp-syntax`

Output Greenplum Database syntax in the `CREATE TABLE` statements. This allows the distribution policy (`DISTRIBUTED BY` or `DISTRIBUTED RANDOMLY` clauses) of a Greenplum Database table to be dumped, which is useful for restoring into other Greenplum Database systems.

`--no-gp-syntax`

Do not output the table distribution clauses in the `CREATE TABLE` statements.

`-? | --help`

Show help about `pg_dumpall` command line arguments, and exit.

Connection Options

`-d` *`connstr`* `|` `--dbname=`*`connstr`*

Specifies parameters used to connect to the server, as a connection string. See *Connection Strings* in the PostgreSQL documentation for more information.

The option is called `--dbname` for consistency with other client applications, but because `pg_dumpall` needs to connect to many databases, the database name in the connection string will be ignored. Use the `-l` option to specify the name of the database used to dump global objects and to discover what other databases should be dumped.

`-h` *`host`* `|` `--host=`*`host`*

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable PGHOST or defaults to localhost.

-l *dbname* | --database=*dbname*

Specifies the name of the database in which to connect to dump global objects. If not specified, the postgres database is used. If the postgres database does not exist, the template1 database is used.

-p *port* | --port=*port*

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable PGPORT or defaults to 5432.

-U *username* | --username= *username*

The database role name to connect as. If not specified, reads from the environment variable PGUSER or defaults to the current system role name.

-w | --no-password

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a .pgpass file the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

-W | --password

Force a password prompt.

--role=*rolename*

Specifies a role name to be used to create the dump. This option causes pg_dumpall to issue a SET ROLE *rolename* command after connecting to the database. It is useful when the authenticated user (specified by -U) lacks privileges needed by pg_dumpall, but can switch to a role with the required rights. Some installations have a policy against logging in directly as a superuser, and use of this option allows dumps to be made without violating the policy.

Notes

Since pg_dumpall calls *pg_dump* internally, some diagnostic messages will refer to pg_dump.

Once restored, it is wise to run ANALYZE on each database so the query planner has useful statistics. You can also run vacuumdb -a -z to analyze all databases.

pg_dumpall requires all needed tablespace directories to exist before the restore; otherwise, database creation will fail for databases in non-default locations.

Examples

To dump all databases:

```
pg_dumpall > db.out
```

To reload database(s) from this file, you can use:

```
psql template1 -f db.out
```

To dump only global objects (including resource queues):

```
pg_dumpall -g --resource-queues
```

See Also

pg_dump

pg_restore

Restores a database from an archive file created by pg_dump.

Synopsis

```
pg_restore [connection-option ...] [restore_option ...] filename

pg_restore -? | --help

pg_restore -V | --version
```

Description

pg_restore is a utility for restoring a database from an archive created by *pg_dump* in one of the non-plain-text formats. It will issue the commands necessary to reconstruct the database to the state it was in at the time it was saved. The archive files also allow pg_restore to be selective about what is restored, or even to reorder the items prior to being restored.

pg_restore can operate in two modes. If a database name is specified, the archive is restored directly into the database. Otherwise, a script containing the SQL commands necessary to rebuild the database is created and written to a file or standard output. The script output is equivalent to the plain text output format of pg_dump. Some of the options controlling the output are therefore analogous to pg_dump options.

pg_restore cannot restore information that is not present in the archive file. For instance, if the archive was made using the "dump data as INSERT commands" option, pg_restore will not be able to load the data using COPY statements.

Options

filename

Specifies the location of the archive file (or directory, for a directory-format archive) to be restored. If not specified, the standard input is used.

Restore Options

-a | --data-only

Restore only the data, not the schema (data definitions). Table data and sequence values are restored, if present in the archive.

This option is similar to, but for historical reasons not identical to, specifying --section=data.

-c | --clean

Clean (drop) database objects before recreating them. (This might generate some harmless error messages, if any objects were not present in the destination database.)

-C | --create

Create the database before restoring into it. If --clean is also specified, drop and recreate the target database before connecting to it.

When this option is used, the database named with -d is used only to issue the initial DROP DATABASE and CREATE DATABASE commands. All data is restored into the database name that appears in the archive.

-d *dbname* | --dbname=*dbname*

Connect to this database and restore directly into this database. This utility, like most other Greenplum Database utilities, also uses the environment variables supported by `libpq`. However it does not read `PGDATABASE` when a database name is not supplied.

`-e | --exit-on-error`

Exit if an error is encountered while sending SQL commands to the database. The default is to continue and to display a count of errors at the end of the restoration.

`-f` *`outfilename`* `| --file=`*`outfilename`*

Specify output file for generated script, or for the listing when used with `-l`. Default is the standard output.

`-F c|d|t | --format={custom | directory | tar}`

The format of the archive produced by *`pg_dump`*. It is not necessary to specify the format, since `pg_restore` will determine the format automatically. Format can be `custom`, `directory`, or `tar`.

`-I` *`index`* `| --index=`*`index`*

Restore definition of named index only.

`-j | --number-of-jobs | --jobs=`*`number-of-jobs`*

Run the most time-consuming parts of `pg_restore` — those which load data, create indexes, or create constraints — using multiple concurrent jobs. This option can dramatically reduce the time to restore a large database to a server running on a multiprocessor machine.

Each job is one process or one thread, depending on the operating system, and uses a separate connection to the server.

The optimal value for this option depends on the hardware setup of the server, of the client, and of the network. Factors include the number of CPU cores and the disk setup. A good place to start is the number of CPU cores on the server, but values larger than that can also lead to faster restore times in many cases. Of course, values that are too high will lead to decreased performance because of thrashing.

Only the custom archive format is supported with this option. The input file must be a regular file (not, for example, a pipe). This option is ignored when emitting a script rather than connecting directly to a database server. Also, multiple jobs cannot be used together with the option `--single-transaction`.

`-l | --list`

List the contents of the archive. The output of this operation can be used with the `-L` option to restrict and reorder the items that are restored.

`-L` *`list-file`* `| --use-list=`*`list-file`*

Restore elements in the *list-file* only, and in the order they appear in the file. Note that if filtering switches such as `-n` or `-t` are used with `-L`, they will further restrict the items restored.

list-file is normally created by editing the output of a previous `-l` operation. Lines can be moved or removed, and can also be commented out by placing a semicolon (;) at the start of the line. See below for examples.

`-n` *`schema`* `| --schema=`*`schema`*

Restore only objects that are in the named schema. This can be combined with the `-t` option to restore just a specific table.

`-O | --no-owner`

Do not output commands to set ownership of objects to match the original database. By default, `pg_restore` issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created schema elements. These statements will fail

unless the initial connection to the database is made by a superuser (or the same user that owns all of the objects in the script). With `-O`, any user name can be used for the initial connection, and this user will own all the created objects.

`-P '`*`function-name`*`(`*`argtype`*` [, ...])' | --function='`*`function-name`*`(`*`argtype`*` [, ...])'`

Restore the named function only. The function name must be enclosed in quotes. Be careful to spell the function name and arguments exactly as they appear in the dump file's table of contents (as shown by the `--list` option).

`-s | --schema-only`

Restore only the schema (data definitions), not data, to the extent that schema entries are present in the archive.

This option is the inverse of `--data-only`. It is similar to, but for historical reasons not identical to, specifying `--section=pre-data --section=post-data`.

(Do not confuse this with the `--schema` option, which uses the word "schema" in a different meaning.)

`-S `*`username`*` | --superuser=`*`username`*

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used.

Note: Greenplum Database does not support user-defined triggers.

`-t `*`table`*` | --table=`*`table`*

Restore definition and/or data of named table only. Multiple tables may be specified with multiple `-t` switches. This can be combined with the `-n` option to specify a schema.

`-T `*`trigger`*` | --trigger=`*`trigger`*

Restore named trigger only.

Note: Greenplum Database does not support user-defined triggers.

`-v | --verbose`

Specifies verbose mode.

`-V | --version`

Print the `pg_restore` version and exit.

`-x | --no-privileges | --no-acl`

Prevent restoration of access privileges (`GRANT/REVOKE` commands).

`-1 | --single-transaction`

Execute the restore as a single transaction. This ensures that either all the commands complete successfully, or no changes are applied.

`--disable-triggers`

This option is relevant only when performing a data-only restore. It instructs `pg_restore` to execute commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have triggers on the tables that you do not want to invoke during data reload. The commands emitted for `--disable-triggers` must be done as superuser. So you should also specify a superuser name with `-S` or, preferably, run `pg_restore` as a superuser.

Note: Greenplum Database does not support user-defined triggers.

`--no-data-for-failed-tables`

By default, table data is restored even if the creation command for the table failed (e.g., because it already exists). With this option, data for such a table is skipped. This behavior is useful when the target database may already contain the desired table contents.

Specifying this option prevents duplicate or obsolete data from being loaded. This option is effective only when restoring directly into a database, not when producing SQL script output.

--no-security-labels

Do not output commands to restore security labels, even if the archive contains them.

--no-tablespaces

Do not output commands to select tablespaces. With this option, all objects will be created in whichever tablespace is the default during restore.

--section=*sectionname*

Only restore the named section. The section name can be `pre-data`, `data`, or `post-data`. This option can be specified more than once to select multiple sections.

The default is to restore all sections.

--use-set-session-authorization

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards-compatible, but depending on the history of the objects in the dump, it might not restore properly.

-? | --help

Show help about `pg_restore` command line arguments, and exit.

Connection Options

-h *host* | --host *host*

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

-p *port* | --port *port*

The TCP port on which the Greenplum Database master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-w | --no-password

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

-W | --password

Force a password prompt.

--role=*rolename*

Specifies a role name to be used to perform the restore. This option causes `pg_restore` to issue a `SET ROLE` *`rolename`* command after connecting to the database. It is useful when the authenticated user (specified by `-U`) lacks privileges needed by `pg_restore`, but can switch to a role with the required rights. Some installations have a policy against logging in directly as a superuser, and use of this option allows restores to be performed without violating the policy.

Notes

If your installation has any local additions to the `template1` database, be careful to load the output of `pg_restore` into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from `template0` not `template1`, for example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

When restoring data to a pre-existing table and the option `--disable-triggers` is used, `pg_restore` emits commands to disable triggers on user tables before inserting the data, then emits commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs may be left in the wrong state.

See also the `pg_dump` documentation for details on limitations of `pg_dump`.

Once restored, it is wise to run `ANALYZE` on each restored table so the query planner has useful statistics.

Examples

Assume we have dumped a database called `mydb` into a custom-format dump file:

```
pg_dump -Fc mydb > db.dump
```

To drop the database and recreate it from the dump:

```
dropdb mydb
pg_restore -C -d template1 db.dump
```

To reload the dump into a new database called `newdb`. Notice there is no `-C`, we instead connect directly to the database to be restored into. Also note that we clone the new database from `template0` not `template1`, to ensure it is initially empty:

```
createdb -T template0 newdb
pg_restore -d newdb db.dump
```

To reorder database items, it is first necessary to dump the table of contents of the archive:

```
pg_restore -l db.dump > db.list
```

The listing file consists of a header and one line for each item, for example,

```
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha2
```

Semicolons start a comment, and the numbers at the start of lines refer to the internal archive ID assigned to each item. Lines in the file can be commented out, deleted, and reordered. For example:

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

Could be used as input to `pg_restore` and would only restore items 10 and 6, in that order:

```
pg_restore -L db.list db.dump
```

See Also

pg_dump

pgbouncer

Manages database connection pools.

Synopsis

```
pgbouncer [OPTION ...] pgbouncer.ini

  OPTION
   [ -d | --daemon ]
   [ -R | --restart ]
   [ -q | --quiet ]
   [ -v | --verbose ]
   [ {-u | --user}=username ]

pgbouncer [ -V | --version ] | [ -h | --help ]
```

Description

PgBouncer is a light-weight connection pool manager for Greenplum and PostgreSQL databases. PgBouncer maintains a pool of connections for each database user and database combination. PgBouncer either creates a new database connection for the client or reuses an existing pooled connection for the same user and database. When the client disconnects, PgBouncer returns the connection to the pool for re-use.

PgBouncer supports the standard connection interface shared by PostgreSQL and Greenplum Database. The Greenplum Database client application (for example, `psql`) should connect to the host and port on which PgBouncer is running rather than directly to the Greenplum Database master host and port.

You configure PgBouncer and its access to Greenplum Database via a configuration file. You provide the configuration file name, usually `pgbouncer.ini`, when you run the `pgbouncer` command. This file provides location information for Greenplum databases. The `pgbouncer.ini` file also specifies process, connection pool, authorized users, and authentication configuration for PgBouncer, among other configuration options.

By default, the `pgbouncer` process runs as a foreground process. You can optionally start `pgbouncer` as a background (daemon) process with the `-d` option.

The `pgbouncer` process is owned by the operating system user that starts the process. You can optionally specify a different user name under which to start `pgbouncer`.

PgBouncer includes a `psql`-like administration console. Authorized users can connect to a virtual database to monitor and manage PgBouncer. You can manage a PgBouncer daemon process via the

admin console. You can also use the console to update and reload the PgBouncer configuration at runtime without stopping and restarting the process.

For additional information about PgBouncer, refer to the *PgBouncer FAQ*.

Options

`-d | --daemon`

Run PgBouncer as a daemon (a background process). The default start-up mode is to run as a foreground process.

When run as a daemon, PgBouncer displays start-up messages to `stdout`. To suppress the display of these messages, include the `-q` option when you start PgBouncer.

To stop a PgBouncer process that was started as a daemon, issue the `SHUTDOWN` command from the PgBouncer administration console.

`-R | --restart`

Restart PgBouncer using the specified command line arguments. Non-TLS connections to databases are maintained during restart; TLS connections are dropped.

To restart PgBouncer as a daemon, specify the options `-Rd`.

> **Note:** Restart is available only if the operating system supports Unix sockets and the PgBouncer `unix_socket_dir` configuration is not disabled.

`-q | --quiet`

Run quietly. Do not display messages to `stdout`.

`-v | --verbose`

Increase message verbosity. Can be specified multiple times.

`{-u | --user}=`*`username`*

Assume the identity of *username* on PgBouncer process start-up.

`-V | --version`

Show the version and exit.

`-h | --help`

Show the command help message and exit.

See Also

`pgbouncer.ini`, *`pgbouncer-admin`*

pgbouncer.ini

PgBouncer configuration file.

Synopsis

```
[databases]
db = ...

[pgbouncer]
...

[users]
...
```

Description

You specify PgBouncer configuration parameters and identify user-specific configuration parameters in the `pgbouncer.ini` configuration file.

The PgBouncer configuration file (typically named `pgbouncer.ini`) is specified in `.ini` format. Files in `.ini` format are composed of sections, parameters, and values. Section names are enclosed in square brackets, for example, `[section_name]`. Parameters and values are specified in *`key=value`* format. Lines beginning with a semicolon (`;`) or pound sign (`#`) are considered comment lines and are ignored.

The PgBouncer configuration file can contain `%include` directives, which specify another file to read and process. This enables you to split the configuration file into separate parts. For example:

```
%include filename
```

If the filename provided is not an absolute path, the file system location is taken as relative to the current working directory.

The PgBouncer configuration file includes the following sections, described in detail below:

- *[databases] Section*
- *[pgbouncer] Section*
- *[users] Section*

[databases] Section

The `[databases]` section contains *`key=value`* pairs, where the *`key`* is a database name and the *`value`* is a `libpq` connect-string list of *`key=value`* pairs.

A database name can contain characters `[0-9A-Za-z_.-]` without quoting. Names that contain other characters must be quoted with standard SQL identifier quoting:

- Enclose names in double quotes (`" "`).
- Represent a double-quote within an identifier with two consecutive double quote characters.

The database name `*` is the fallback database. PgBouncer uses the value for this key as a connect string for the requested database. Automatically-created database entries such as these are cleaned up if they remain idle longer than the time specified in `autodb_idle_timeout` parameter.

Database Connection Parameters

The following parameters may be included in the *`value`* to specify the location of the database.

`dbname`

The destination database name.

Default: the client-specified database name

`host`

The name or IP address of the Greenplum master host. Host names are resolved at connect time. If DNS returns several results, they are used in a round-robin manner. The DNS result is cached and the `dns_max_ttl` parameter determines when the cache entry expires.

Default: not set; the connection is made through a Unix socket

`port`

The Greenplum Database master port.

Default: 5432

`user, password`

If `user=` is set, all connections to the destination database are initiated as the specified user, resulting in a single connection pool for the database.

If the `user=` parameter is not set, PgBouncer attempts to log in to the destination database with the user name passed by the client. In this situation, there will be one pool for each user who connects to the database.

`auth_user`

If `auth_user` is set, any user who is not specified in `auth_file` is authenticated by querying the `pg_shadow` database view. PgBouncer performs this query as the `auth_user` Greenplum Database user. `auth_user`'s password must be set in the `auth_file`.

`client_encoding`

Ask for specific `client_encoding` from server.

`datestyle`

Ask for specific `datestyle` from server.

`timezone`

Ask for specific `timezone` from server.

Pool Configuration

You can use the following parameters for database-specific pool configuration.

`pool_size`

Set the maximum size of pools for this database. If not set, the `default_pool_size` is used.

`connect_query`

Query to be executed after a connection is established, but before allowing the connection to be used by any clients. If the query raises errors, they are logged but ignored otherwise.

`pool_mode`

Set the pool mode for this database. If not set, the default `pool_mode` is used.

`max_db_connections`

Set a database-wide maximum number of PgBouncer connections for this database. The total number of connections for all pools for this database will not exceed this value.

[pgbouncer] Section

Generic Settings

`logfile`

The location of the log file. The log file is kept open. After log rotation, execute `kill -HUP pgbouncer` or run the `RELOAD;` command in the PgBouncer Administration Console.

Default: not set

`pidfile`

The name of the pid file. Without a pidfile, you cannot run PgBouncer as a background (daemon) process.

Default: not set

`listen_addr`

A list of interface addresses where PgBouncer listens for TCP connections. You may also use `*`, which means to listen on all interfaces. If not set, only Unix socket connections are allowed.

Specify addresses numerically (IPv4/IPv6) or by name.

Default: not set

listen_port

The port PgBouncer listens on. Applies to both TCP and Unix sockets.

Default: 6432

unix_socket_dir

Specifies the location for the Unix sockets. Applies to both listening socket and server connections. If set to an empty string, Unix sockets are disabled. Required for online restart (`-R` option) to work.

Default: `/tmp`

unix_socket_mode

Filesystem mode for the Unix socket.

Default: 0777

unix_socket_group

Group name to use for Unix socket.

Default: not set

user

If set, specifies the Unix user to change to after startup. This works only if PgBouncer is started as root or if `user` is the same as the current user.

Default: not set

auth_file

The name of the file containing the user names and passwords to load. The file format is the same as the Greenplum Database `pg_auth` file.

Default: not set

auth_hba_file

HBA configuration file to use when `auth_type` is `hba`. Refer to the *HBA file format* discussion in the PgBouncer documentation for information about PgBouncer support of the HBA authentication file format.

Default: not set

auth_type

How to authenticate users.

pam

Use PAM to authenticate users. `auth_file` is ignored. This method is not compatible with databases using the `auth_user` option. The service name reported to PAM is “pgbouncer”. PAM is not supported in the HBA configuration file.

hba

The actual authentication type is loaded from the `auth_hba_file`. This setting allows different authentication methods different access paths.

cert

Clients must connect with TLS using a valid client certificate. The client's username is taken from CommonName field in the certificate.

md5

Use MD5-based password check. `auth_file` may contain both MD5-encrypted or plain-text passwords. This is the default authentication method.

plain

Clear-text password is sent over wire. *Deprecated*.

trust

No authentication is performed. The username must still exist in the `auth_file`.

any

Like the `trust` method, but the username supplied is ignored. Requires that all databases are configured to log in with a specific user. Additionally, the console database allows any user to log in as admin.

auth_query

Query to load a user's password from the database. If a user does not exist in the `auth_file` and the database entry includes an `auth_user`, this query is run in the database as `auth_user` to lookup up the user.

Note that the query is run inside target database, so if a function is used it needs to be installed into each database.

Default: `SELECT usename, passwd FROM pg_shadow WHERE usename=$1`

auth_user

If `auth_user` is set, any user who is not specified in `auth_file` is authenticated through the `auth_query` query from the `pg_shadow` database view. PgBouncer performs this query as the `auth_user` Greenplum Database user. `auth_user`'s password must be set in the `auth_file`.

Direct access to `pg_shadow` requires Greenplum Database administrative privileges. It is preferable to use a non-admin user that calls `SECURITY DEFINER` function instead.

pool_mode

Specifies when a server connection can be reused by other clients.

session

Connection is returned to the pool when the client disconnects. Default.

transaction

Connection is returned to the pool when the transaction finishes.

statement

Connection is returned to the pool when the current query finishes. Long transactions with multiple statements are disallowed in this mode.

max_client_conn

Maximum number of client connections allowed. When increased, you should also increase the file descriptor limits. The actual number of file descriptors used is more than `max_client_conn`. The theoretical maximum used, when each user connects with its own username to the server is:

```
max_client_conn + (max_pool_size * total_databases * total_users)
```

If a database user is specified in the connect string, all users connect using the same username. Then the theoretical maximum connections is:

```
max_client_conn + (max_pool_size * total_databases)
```

(The theoretical maximum should be never reached, unless someone deliberately crafts a load for it. Still, it means you should set the number of file descriptors to a safely high number. Search for `ulimit` in your operating system documentation.)

Default: 100

default_pool_size

The number of server connections to allow per user/database pair. This can be overridden in the per-database configuration.

Default: 20

min_pool_size

Add more server connections to the pool when it is lower than this number. This improves behavior when the usual load drops and then returns suddenly after a period of total inactivity.

Default: 0 (disabled)

reserve_pool_size

The number of additional connections to allow for a pool. 0 disables.

Default: 0 (disabled)

reserve_pool_timeout

If a client has not been serviced in this many seconds, PgBouncer enables use of additional connections from the reserve pool. 0 disables.

Default: 5.0

max_db_connections

The maximum number of connections per database. If you hit the limit, closing a client connection to one pool does not immediately allow a server connection to be established for another pool, because the server connection for the first pool is still open. Once the server connection closes (due to idle timeout), a new server connection will be opened for the waiting pool.

Default: unlimited

max_user_connections

The maximum number of connections per-user. When you hit the limit, closing a client connection to one pool does not immediately allow a connection to be established for another pool, because the connection for the first pool is still open. After the connection for the first pool has closed (due to idle timeout), a new server connection is opened for the waiting pool.

server_round_robin

By default, PgBouncer reuses server connections in LIFO (last-in, first-out) order, so that a few connections get the most load. This provides the best performance when a single server serves a database. But if there is TCP round-robin behind a database IP, then it is better if PgBouncer also uses connections in that manner to achieve uniform load.

Default: 0

ignore_startup_parameters

By default, PgBouncer allows only parameters it can keep track of in startup packets: `client_encoding`, `datestyle`, `timezone`, and `standard_conforming_strings`.

All others parameters raise an error. To allow other parameters, specify them here so that PgBouncer can ignore them.

Default: empty

disable_pqexec

Disable Simple Query protocol (PQexec). Unlike Extended Query protocol, Simple Query protocol allows multiple queries in one packet, which allows some classes of SQL-injection attacks. Disabling it can improve security. This means that only clients that exclusively use Extended Query protocol will work.

Default: 0

application_name_add_host

Add the client host address and port to the application name setting set on connection start. This helps in identifying the source of bad queries. The setting is overwritten without detection if the application executes `SET application_name` after connecting.

Default: 0

conffile

Show location of the current configuration file. Changing this parameter will result in PgBouncer using another config file for next RELOAD / SIGHUP.

Default: file from command line

service_name

Used during win32 service registration.

Default: pgbouncer

job_name

Alias for `service_name`.

Log Settings

syslog

Toggles syslog on and off.

Default: 0

syslog_ident

Under what name to send logs to syslog.

Default: `pgbouncer`

syslog_facility

Under what facility to send logs to syslog. Some possibilities are: `auth`, `authpriv`, `daemon`, `user`, `local0-7`

Default: `daemon`

log_connections

Log successful logins.

Default: 1

log_disconnections

Log disconnections, with reasons.

Default: 1

log_pooler_errors

Log error messages that the pooler sends to clients.

Default: 1

log_stats

Write aggregated statistics into the log, every `stats_period`. This can be disabled if external monitoring tools are used to grab the same data from `SHOW` commands.

Default: 1

stats_period

How often to write aggregated statistics to the log.

Default: 60

Console Access Control

admin_users

Comma-separated list of database users that are allowed to connect and run all commands on the PgBouncer Administration Console. Ignored when `auth_type=any`, in which case any username is allowed in as admin.

Default: empty

stats_users

Comma-separated list of database users that are allowed to connect and run read-only queries on the console. This includes all `SHOW` commands except `SHOW FDS`.

Default: empty

Connection Sanity Checks, Timeouts

server_reset_query

Query sent to server on connection release, before making it available to other clients. At that moment no transaction is in progress so it should not include `ABORT` or `ROLLBACK`.

The query should clean any changes made to a database session so that the next client gets a connection in a well-defined state. Default is `DISCARD ALL` which cleans everything, but that leaves the next client no pre-cached state.

Note: Greenplum Database does not support `DISCARD ALL`.

You can use other commands to clean up the session state. For example, `DEALLOCATE ALL` drops prepared statements, and `DISCARD TEMP` drops temporary tables.

When transaction pooling is used, the `server_reset_query` should be empty, as clients should not use any session features. If clients do use session features, they will be broken because transaction pooling does not guarantee that the next query will run on the same connection.

Default: `DISCARD ALL;` (Not supported by Greenplum Database.)

server_reset_query_always

Whether `server_reset_query` should be run in all pooling modes. When this setting is off (default), the `server_reset_query` will be run only in pools that are in sessions pooling mode. Connections in transaction pooling mode should not have any need for reset query.

Default: 0

server_check_delay

How long to keep released connections available for re-use without running sanity-check queries on it. If 0, then the query is run always.

Default: 30.0

server_check_query

A simple do-nothing query to test the server connection.

If an empty string, then sanity checking is disabled.

Default: SELECT 1;

server_fast_close

Disconnect a server in session pooling mode immediately or after the end of the current transaction if it is in “close_needed” mode (set by `RECONNECT`, `RELOAD` that changes

connection settings, or DNS change), rather than waiting for the session end. In statement or transaction pooling mode, this has no effect since that is the default behavior there.

If because of this setting a server connection is closed before the end of the client session, the client connection is also closed. This ensures that the client notices that the session has been interrupted.

This setting makes connection configuration changes take effect sooner if session pooling and long-running sessions are used. The downside is that client sessions are liable to be interrupted by a configuration change, so client applications will need logic to reconnect and reestablish session state. But note that no transactions will be lost, because running transactions are not interrupted, only idle sessions.

Default: 0

`server_lifetime`

The pooler tries to close server connections that have been connected longer than this number of seconds. Setting it to 0 means the connection is to be used only once, then closed.

Default: 3600.0

`server_idle_timeout`

If a server connection has been idle more than this many seconds it is dropped. If this parameter is set to 0, timeout is disabled. [seconds]

Default: 600.0

`server_connect_timeout`

If connection and login will not finish in this number of seconds, the connection will be closed.

Default: 15.0

`server_login_retry`

If a login fails due to failure from `connect()` or authentication, the pooler waits this many seconds before retrying to connect.

Default: 15.0

`client_login_timeout`

If a client connects but does not manage to login in this number of seconds, it is disconnected. This is needed to avoid dead connections stalling `SUSPEND` and thus online restart.

Default: 60.0

`autodb_idle_timeout`

If database pools created automatically (via `*`) have been unused this many seconds, they are freed. Their statistics are also forgotten.

Default: 3600.0

`dns_max_ttl`

How long to cache DNS lookups, in seconds. If a DNS lookup returns several answers, PgBouncer round-robins between them in the meantime. The actual DNS TTL is ignored.

Default: 15.0

`dns_nxdomain_ttl`

How long error and NXDOMAIN DNS lookups can be cached, in seconds.

Default: 15.0

dns_zone_check_period

Period to check if zone serial numbers have changed.

PgBouncer can collect DNS zones from hostnames (everything after first dot) and then periodically check if the zone serial numbers change. If changes are detected, all hostnames in that zone are looked up again. If any host IP changes, its connections are invalidated.

Works only with UDNS and c-ares backend (`--with-udns` or `--with-cares` to configure).

Default: 0.0 (disabled)

TLS settings

client_tls_sslmode

TLS mode to use for connections from clients. TLS connections are disabled by default. When enabled, `client_tls_key_file` and `client_tls_cert_file` must be also configured to set up the key and certificate PgBouncer uses to accept client connections.

- `disable`: Plain TCP. If client requests TLS, it's ignored. Default.
- `allow`: If client requests TLS, it is used. If not, plain TCP is used. If client uses client-certificate, it is not validated.
- `prefer`: Same as `allow`.
- `require`: Client must use TLS. If not, client connection is rejected. If client uses client-certificate, it is not validated.
- `verify-ca`: Client must use TLS with valid client certificate.
- `verify-full`: Same as `verify-ca`.

client_tls_key_file

Private key for PgBouncer to accept client connections.

Default: not set

client_tls_cert_file

Root certificate file to validate client certificates.

Default: unset

client_tls_ca_file

Root certificate to validate client certificates.

Default: unset

client_tls_protocols

Which TLS protocol versions are allowed.

Valid values: are `tlsv1.0`, `tlsv1.1`, `tlsv1.2`.

Shortcuts: `all` (`tlsv1.0`,`tlsv1.1`,`tlsv1.2`), `secure` (`tlsv1.2`), `legacy` (`all`).

Default: `secure`

client_tls_ciphers

Default: `fast`

client_tls_ecdhcurve

Elliptic Curve name to use for ECDH key exchanges.

Allowed values: `none` (DH is disabled), `auto` (256-bit ECDH), curve name.

Default: `auto`

client_tls_dheparams

DHE key exchange type.

Allowed values: `none` (DH is disabled), `auto` (2048-bit DH), `legacy` (1024-bit DH).

Default: `auto`

`server_tls_sslmode`

TLS mode to use for connections to Greenplum Database and PostgreSQL servers. TLS connections are disabled by default.

- `disabled`: Plain TCP. TLS is not requested from the server. Default.
- `allow`: If server rejects plain, try TLS. (*PgBouncer Documentation is speculative on this..*)
- `prefer`: TLS connection is always requested first. When connection is refused, plain TPC is used. Server certificate is not validated.
- `require`: Connection must use TLS. If server rejects it, plain TCP is not attempted. Server certificate is not validated.
- `verify-ca`: Connection must use TLS and server certificate must be valid according to `server_tls_ca_file`. The server hostname is not verfied against the certificate.
- `verify-full`: Connection must use TLS and the server certificate must be valid according to `server_tls_ca_file`. The server hostname must match the hostname in the certificate.

`server_tls_ca_file`

Path to the root certificate file used to validate Greenplum Database and PostgreSQL server certificates.

Default: unset

`server_tls_key_file`

The private key for PgBouncer to authenticate against Greenplum Database or PostgreSQL server.

Default: not set

`server_tls_cert_file`

Certificate for private key. Greenplum Database or PostgreSQL servers can validate it.

Default: not set

`server_tls_protocols`

Which TLS protocol versions are allowed.

Valid values are: `tlsv1.0`, `tlsv1.1`, `tlsv1.2`.

Shortcuts: `all` (`tlsv1.0`, `tlsv1.1`, `tlsv1.2`); `secure` (`tlsv1.2`); `legacy` (`all`).

Default: `secure`

`server_tls_ciphers`

Default: `fast`

Dangerous Timeouts

Setting the following timeouts can cause unexpected errors.

`query_timeout`

Queries running longer than this (seconds) are canceled. This parameter should be used only with a slightly smaller server-side `statement_timeout`, to trap queries with network problems. [seconds]

Default: 0.0 (disabled)

query_wait_timeout

The maximum time, in seconds, queries are allowed to wait for execution. If the query is not assigned a connection during that time, the client is disconnected. This is used to prevent unresponsive servers from grabbing up connections.

Default: 120

client_idle_timeout

Client connections idling longer than this many seconds are closed. This should be larger than the client-side connection lifetime settings, and only used for network problems.

Default: 0.0 (disabled)

idle_transaction_timeout

If client has been in "idle in transaction" state longer than this (seconds), it is disconnected.

Default: 0.0 (disabled)

Low-level Network Settings

pkt_buf

Internal buffer size for packets. Affects the size of TCP packets sent and general memory usage. Actual `libpq` packets can be larger than this so there is no need to set it large.

Default: 4096

max_packet_size

Maximum size for packets that PgBouncer accepts. One packet is either one query or one result set row. A full result set can be larger.

Default: 2147483647

listen_backlog

Backlog argument for the `listen(2)` system call. It determines how many new unanswered connection attempts are kept in queue. When the queue is full, further new connection attempts are dropped.

Default: 128

sbuf_loopcnt

How many times to process data on one connection, before proceeding. Without this limit, one connection with a big result set can stall PgBouncer for a long time. One loop processes one `pkt_buf` amount of data. 0 means no limit.

Default: 5

SO_REUSEPORT

Specifies whether to set the socket option `SO_REUSEPORT` on TCP listening sockets. On some operating systems, this allows running multiple PgBouncer instances on the same host listening on the same port and having the kernel distribute the connections automatically. This option is a way to get PgBouncer to use more CPU cores. (PgBouncer is single-threaded and uses one CPU core per instance.)

The behavior in detail depends on the operating system kernel. As of this writing, this setting has the desired effect on recent versions of Linux. On systems that don't support the socket option at all, turning this setting on will result in an error.

Each PgBouncer instance on the same host needs different settings for at least `unix_socket_dir` and `pidfile`, as well as `logfile` if that is used. Also note that if you make use of this option, you can no longer connect to a specific PgBouncer instance via TCP/IP, which might have implications for monitoring and metrics collection.

Default: 0

suspend_timeout

How many seconds to wait for buffer flush during SUSPEND or restart (-R). Connection is dropped if flush does not succeed.

Default: 10

tcp_defer_accept

For details on this and other TCP options, please see the tcp(7) man page.

Default: 45 on Linux, otherwise 0

tcp_socket_buffer

Default: not set

tcp_keepalive

Turns on basic keepalive with OS defaults.

On Linux, the system defaults are tcp_keepidle=7200, tcp_keepintvl=75, tcp_keepcnt=9.

Default: 1

tcp_keepcnt

Default: not set

tcp_keepidle

Default: not set

tcp_keepintvl

Default: not set

tcp_user_timeout

Sets the TCP_USER_TIMEOUT socket option. This specifies the maximum amount of time in milliseconds that transmitted data may remain unacknowledged before the TCP connection is forcibly closed. If set to 0, then operating system's default is used.

Default: 0

[users] Section

This section contains *key*=*value* pairs, where the *key* is a user name and the *value* is a libpq connect-string list of *key*=*value* pairs.

Pool configuration

pool_mode

Set the pool mode for all connections from this user. If not set, the database or default pool_mode is used.

Example Configuration Files

Minimal Configuration

```
[databases]
postgres = host=127.0.0.1 dbname=postgres auth_user=gpadmin

[pgbouncer]
pool_mode = session
listen_port = 6543
listen_addr = 127.0.0.1
auth_type = md5
```

```
auth_file = users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = someuser
stats_users = stat_collector
```

Use connection parameters passed by the client:

```
[databases]
* =

[pgbouncer]
listen_port = 6543
listen_addr = 0.0.0.0
auth_type = trust
auth_file = bouncer/users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
ignore_startup_parameters=options
```

Database Defaults

```
[databases]

; foodb over unix socket
foodb =

; redirect bardb to bazdb on localhost
bardb = host=127.0.0.1 dbname=bazdb

; access to destination database will go with single user
forcedb = host=127.0.0.1 port=300 user=baz password=foo
 client_encoding=UNICODE datestyle=ISO
```

See Also

pgbouncer, *pgbouncer-admin*, *PgBouncer Configuration Page*

pgbouncer-admin

PgBouncer Administration Console.

Synopsis

```
psql -p port pgbouncer
```

Description

The PgBouncer Administration Console is available via `psql`. Connect to the PgBouncer *port* and the virtual database named `pgbouncer` to log in to the console.

Users listed in the `pgbouncer.ini` configuration parameters `admin_users` and `stats_users` have privileges to log in to the PgBouncer Administration Console.

You can control connections between PgBouncer and Greenplum Database from the console. You can also set PgBouncer configuration parameters.

Options

-p *port*

The PgBouncer port number.

Command Syntax

```
pgbouncer=# SHOW help;
NOTICE:  Console usage
DETAIL:
    SHOW HELP|CONFIG|DATABASES|POOLS|CLIENTS|SERVERS|VERSION
    SHOW FDS|SOCKETS|ACTIVE_SOCKETS|LISTS|MEM
    SHOW DNS_HOSTS|DNS_ZONES
    SHOW STATS|STATS_TOTALS|STATS_AVERAGES
    SET key = arg
    RELOAD
    PAUSE [<db>]
    RESUME [<db>]
    DISABLE <db>
    ENABLE <db>
    KILL <db>
    SUSPEND
    SHUTDOWN
```

Administration Commands

The following PgBouncer administration commands control the running `pgbouncer` process.

PAUSE [*db*]

If no database is specified, PgBouncer tries to disconnect from all servers, first waiting for all queries to complete. The command will not return before all queries are finished. This command is to be used to prepare to restart the database.

If a database name is specified, PgBouncer pauses only that database.

If you run a `PAUSE` *db* command, and then a `PAUSE` command to pause all databases, you must execute two `RESUME` commands, one for all databases, and one for the named database.

SUSPEND

All socket buffers are flushed and PgBouncer stops listening for data on them. The command will not return before all buffers are empty. To be used when rebooting PgBouncer online.

RESUME [*db*]

Resume work from a previous `PAUSE` or `SUSPEND` command.

If a database was specified for the `PAUSE` command, the database must also be specified with the `RESUME` command.

After pausing all databases with the `PAUSE` command, resuming a single database with `RESUME` *db* is not supported.

DISABLE *db*

Reject all new client connections on the database.

ENABLE *db*

Allow new client connections on the database.

KILL *db*

Immediately drop all client and server connections to the named database.

SHUTDOWN

Stop PgBouncer process. To exit from the `psql` command line session, enter `\q`.

RECONNECT

Close each open server connection for the given database, or all databases, after it is released (according to the pooling mode), even if its lifetime is not up yet. New server connections can be made immediately and will connect as necessary according to the pool size settings.

This command is useful when the server connection setup has changed, for example to perform a gradual switchover to a new server. It is *not* necessary to run this command when the connection string in pgbouncer.ini has been changed and reloaded (see `RELOAD`) or when DNS resolution has changed, because then the equivalent of this command will be run automatically. This command is only necessary if something downstream of PgBouncer routes the connections.

After this command is run, there could be an extended period where some server connections go to an old destination and some server connections go to a new destination. This is likely only sensible when switching read-only traffic between read-only replicas, or when switching between nodes of a multimaster replication setup. If all connections need to be switched at the same time, `PAUSE` is recommended instead. To close server connections without waiting (for example, in emergency failover rather than gradual switchover scenarios), also consider `KILL`.

RELOAD

The PgBouncer process reloads the current configuration file and updates the changeable settings.

WAIT_CLOSE [*db*]

Wait until all server connections, either of the specified database or of all databases, have cleared the "close_needed" state (see `SHOW SERVERS`). This can be called after a `RECONNECT` or `RELOAD` to wait until the respective configuration change has been fully activated, for example in switchover scripts.

SET *key = value*

Override specified configuration setting. See the *`SHOW CONFIG;`* command.

SHOW Command

The `SHOW` *`category`* command displays different types of PgBouncer information. You can specify one of the following categories:

- *ACTIVE_SOCKETS*
- *CLIENTS*
- *CONFIG*
- *DATABASES*
- *DNS_HOSTS*
- *DNS_ZONES*
- *FDS*
- *POOLS*
- *SERVERS*
- *SOCKETS*
- *STATS*
- *STATS_TOTALS*
- *STATS_AVERAGES*
- *LISTS*
- *MEM*
- *USERS*
- *VERSION*

ACTIVE_SOCKETS

Table 69: Active Socket Information

Column	Description
type	S, for server, C for client.
user	Username `pgbouncer` uses to connect to server.
database	Database name.
state	State of the server connection, one of `active`, `used` or `idle`.
addr	IP address of PostgreSQL server.
port	Port of PostgreSQL server.
local_addr	Connection start address on local machine.
local_port	Connection start port on local machine.
connect_time	When the connection was made.
request_time	When last request was issued.
wait	Time waiting.
wait_us	Time waiting (microseconds).
ptr	Address of internal object for this connection. Used as unique ID.
link	Address of client connection the server is paired with.
remote_pid	Process identifier of backend server process.
tls	TLS context.
recv_pos	Receive position in the I/O buffer.
pkt_pos	Parse position in the I/O buffer.
pkt_remain	Number of packets remaining on the socket.
send_pos	Send position in the packet.
send_remain	Total packet length remaining to send.
pkt_avail	Amount of I/O buffer left to parse.
send_avail	Amount of I/O buffer left to send.

CLIENTS

Table 70: Clients

Column	Description
type	C, for client.
user	Client connected user.
database	Database name.
state	State of the client connection, one of `active`, `used`, `waiting` or `idle`.

Column	Description
addr	IP address of client, or `unix` for a socket connection.
port	Port client is connected to.
local_addr	Connection end address on local machine.
local_port	Connection end port on local machine.
connect_time	Timestamp of connect time.
request_time	Timestamp of latest client request.
wait	Time waiting.
wait_us	Time waiting (microseconds).
ptr	Address of internal object for this connection. Used as unique ID.
link	Address of server connection the client is paired with.
remote_pid	Process ID, if client connects with Unix socket and the OS supports getting it.
tls	Client TLS context.

CONFIG

List of current PgBouncer parameter settings

Table 71: Config

Column	Description
key	Configuration variable name
value	Configuration value
changeable	Either `yes` or `no`. Shows whether the variable can be changed while running. If `no`, the variable can be changed only at boot time.

DATABASES

Table 72: Databases

Column	Description
name	Name of configured database entry.
host	Host pgbouncer connects to.
port	Port pgbouncer connects to.
database	Actual database name pgbouncer connects to.
force_user	When user is part of the connection string, the connection between pgbouncer and the database server is forced to the given user, whatever the client user.
pool_size	Maximum number of server connections.
reserve_pool	The number of additional connections that can be created if the pool reaches `pool_size`.

Column	Description
pool_mode	The database's override `pool_mode` or NULL if the default will be used instead.
max_connections	Maximum number of connections for all pools for this database.
current_connections	The total count of connections for all pools for this database.
paused	Paused/unpaused state of the database.
disabled	Enabled/disabled state of the database.

DNS_HOSTS

Table 73: DNS Zones in Cache

Column	Description
hostname	Host name
ttl	How many seconds until next lookup.
addrs	Comma-separated list of addresses.

DNS_ZONES

Table 74: DNS Zones in Cache

Column	Description
zonename	Zone name
serial	Current DNS serial number
count	Hostnames belonging to this zone

FDS

`SHOW FDS` is an internal command used for an online restart, for example when upgrading to a new PgBouncer version. It displays a list of file descriptors in use with the internal state attached to them. This command blocks the internal event loop, so it should not be used while PgBouncer is in use.

When the connected user has username "pgbouncer", connects through a Unix socket, and has the same UID as the running process, the actual file descriptors are passed over the connection.

Table 75: FDS

Column	Description
fd	File descriptor numeric value.
task	One of `pooler`, `client`, or `server`.
user	User of the connection using the file descriptor.
database	Database of the connection using the file descriptor.
addr	IP address of the connection using the file descriptor, "unix" if a Unix socket is used.

Column	Description
port	Port used by the connection using the file descriptor.
cancel	Cancel key for this connection.
link	File descriptor for corresponding server/client. NULL if idle.
client_encoding	Character set used for the database.
std_strings	This controls whether ordinary string literals ('...') treat backslashes literally, as specified in the SQL standard.
datestyle	Display format for date and time values.
timezone	The timezone for interpreting and displaying time stamps.
password	`auth_user`'s password.

LISTS

Shows the following PgBouncer statistcs in two columns: the item label and value.

Table 76: Count of PgBouncer Items

Item	Description
databases	Count of databases.
users	Count of users.
pools	Count of pools.
free_clients	Count of free clients.
used_clients	Count of used clients.
login_clients	Count of clients in `login` state.
free_servers	Count of free servers.
used_servers	Count of used servers.
dns_names	Count of DNS names.
dns_zones	Count of DNS zones.
dns_queries	Count of DNS queries.
dns_pending	Count of in-flight DNS queries.

MEM

Shows cache memory information for these PgBouncer caches:

- user_cache
- db_cache
- pool_cache
- server_cache
- client_cache
- iobuf_cache

Table 77: In Memory Cache

Column	Description
name	Name of cache.
size	The size of a single slot in the cache.
used	Number of used slots in the cache.
free	The number of available slots in the cache.
memtotal	Total bytes used by the cache.

POOLS

A new pool entry is made for each pair of (database, user).

Table 78: Pools

Column	Description
database	Database name.
user	User name.
cl_active	Client connections that are linked to server connection and can process queries.
cl_waiting	Client connections have sent queries but have not yet got a server connection.
sv_active	Server connections that linked to client.
sv_idle	Server connections that are unused and immediately usable for client queries.
sv_used	Server connections that have been idle more than `server_check_delay`. The `server_check_query` query must be run on them before they can be used.
sv_tested	Server connections that are currently running either `server_reset_query` or `server_check_query`.
sv_login	Server connections currently in process of logging in.
maxwait	How long the first (oldest) client in the queue has waited, in seconds. If this begins to increase, the current pool of servers does not handle requests fast enough. The cause may be either an overloaded server or the `pool_size` setting is too small.
maxwait_us	`max_wait` (microseconds).
pool_mode	The pooling mode in use.

SERVERS

Table 79: Servers

Column	Description
type	S, for server.

Column	Description
user	User ID that `pgbouncer` uses to connect to server.
database	Database name.
state	State of the pgbouncer server connection, one of `active`, `used`, or `idle`.
addr	IP address of the Greenplum or PostgreSQL server.
port	Port of the Greenplum or PostgreSQL server.
local_addr	Connection start address on local machine.
local_port	Connection start port on local machine.
connect_time	When the connection was made.
request_time	When the last request was issued.
wait	Time waiting.
wait_us	Time waiting (microseconds).
close_needed	1 if the connection will be closed as soon as possible, because a configuration file reload or DNS update changed the connection information or `RECONNECT` was issued.
ptr	Address of the internal object for this connection. Used as unique ID.
link	Address of gthe client connection the server is paired with.
remote_pid	Pid of backend server process. If the connection is made over Unix socket and the OS supports getting process ID info, it is the OS pid. Otherwise it is extracted from the cancel packet the server sent, which should be PID in case server is PostgreSQL, but it is a random number in case server is another PgBouncer.
tls	TLS context.

STATS

Shows statistics.

Table 80: Stats

Column	Description
database	Statistics are presented per database.
total_xact_count	Total number of SQL transactions pooled by PgBouncer.
total_query_count	Total number of SQL queries pooled by PgBouncer.
total_received	Total volume in bytes of network traffic received by `pgbouncer`.
total_sent	Total volume in bytes of network traffic sent by `pgbouncer`.
total_xact_time	Total number of microseconds spent by PgBouncer when connected to Greenplum Database in a transaction, either idle in transaction or executing queries.
total_query_time	Total number of microseconds spent by `pgbouncer` when actively connected to the database server.
total_wait_time	Time spent (in microseconds) by clients waiting for a server.

Column	Description
avg_xact_count	Average number of SQL transactions pooled by PgBouncer.
avg_query_count	Average queries per second in last stats period.
avg_recv	Average received (from clients) bytes per second.
avg_sent	Average sent (to clients) bytes per second.
avg_xact_time	Average transaction duration in microseconds.
avg_query_time	Average query duration in microseconds.
avg_wait_time	Time spent by clients waiting for a server in microseconds (average per second).

STATS_AVERAGES

Subset of `SHOW STATS` showing the average values for selected statistics.

STATS_TOTALS

Subset of `SHOW STATS` showing the total values for selected statistics.

USERS

Table 81: Users

Column	Description
name	The user name
pool_mode	The user's override pool_mode, or NULL if the default will be used instead.

VERSION

Display PgBouncer version information.

Note: This reference documentation is based on the PgBouncer 1.13 documentation.

See Also

`pgbouncer`, *`pgbouncer.ini`*

plcontainer

The `plcontainer` utility installs Docker images and manages the PL/Container configuration. The utility consists of two sets of commands.

- `image-*` commands manage Docker images on the Greenplum Database system hosts.
- `runtime-*` commands manage the PL/Container configuration file on the Greenplum Database instances. You can add Docker image information to the PL/Container configuration file including the image name, location, and shared folder information. You can also edit the configuration file.

To configure PL/Container to use a Docker image, you install the Docker image on all the Greenplum Database hosts and then add configuration information to the PL/Container configuration.

PL/Container configuration values, such as image names, runtime IDs, and parameter values and names are case sensitive.

plcontainer Syntax

```
plcontainer [command] [-h | --help]  [--verbose]
```

Where *command* is one of the following.

```
  image-add {{-f | --file} image_file [-ulc | --use_local_copy]} | {{-u | --
URL} image_URL}
  image-delete {-i | --image} image_name
  image-list

  runtime-add {-r | --runtime} runtime_id
     {-i | --image} image_name {-l | --language} {python | python3 | r}
     [{-v | --volume} shared_volume [{-v| --volume} shared_volume...]]
     [{-s | --setting} param=value [{-s | --setting} param=value ...]]
  runtime-replace {-r | --runtime} runtime_id
     {-i | --image} image_name -l {r | python}
     [{-v | --volume} shared_volume [{-v | --volume} shared_volume...]]
     [{-s | --setting} param=value [{-s | --setting} param=value ...]]
  runtime-show {-r | --runtime} runtime_id
  runtime-delete {-r | --runtime} runtime_id
  runtime-edit [{-e | --editor} editor]
  runtime-backup {-f | --file} config_file
  runtime-restore {-f | --file} config_file
  runtime-verify
```

plcontainer Commands and Options

image-add *location*

Install a Docker image on the Greenplum Database hosts. Specify either the location of the Docker image file on the host or the URL to the Docker image. These are the supported location options:

- {**-f** | **--file**} *image_file* Specify the file system location of the Docker image tar archive file on the local host. This example specifies an image file in the `gpadmin` user's home directory: `/home/gpadmin/test_image.tar.gz`
- {**-u** | **--URL**} *image_URL* Specify the URL of the Docker repository and image. This example URL points to a local Docker repository `192.168.0.1:5000/images/mytest_plc_r:devel`

By default, the `image-add` command copies the image to each Greenplum Database segment and standby master host, and installs the image. When you specify an *image_file* and provide the [**-ulc** | **--use_local_copy**] option, `plcontainer` installs the image only on the host on which you execute the command.

After installing the Docker image, use the *runtime-add* command to configure PL/Container to use the Docker image.

image-delete {-i | --image} *image_name*

Remove an installed Docker image from all Greenplum Database hosts. Specify the full Docker image name including the tag for example `pivotaldata/plcontainer_python_shared:1.0.0`

image-list

List the Docker images installed on the host. The command list only the images on the local host, not remote hosts. The command lists all installed Docker images, including images installed with Docker commands.

runtime-add *options*

Add configuration information to the PL/Container configuration file on all Greenplum Database hosts. If the specified *runtime_id* exists, the utility returns an error and the configuration information is not added.

These are the supported options:

{-i | --image} *docker-image*

Required. Specify the full Docker image name, including the tag, that is installed on the Greenplum Database hosts. For example `pivotaldata/plcontainer_python:1.0.0`.

The utility returns a warning if the specified Docker image is not installed.

The `plcontainer image-list` command displays installed image information including the name and tag (the Repository and Tag columns).

{-l | --language} python | python3 | r

Required. Specify the PL/Container language type, supported values are `python` (PL/Python using Python 2), `python3` (PL/Python using Python 3) and `r` (PL/R). When adding configuration information for a new runtime, the utility adds a startup command to the configuration based on the language you specify.

Startup command for the Python 2 language.

```
/clientdir/pyclient.sh
```

Startup command for the Python 3 language.

```
/clientdir/pyclient3.sh
```

Startup command for the R language.

```
/clientdir/rclient.sh
```

{-r | --runtime} *runtime_id*

Required. Add the runtime ID. When adding a `runtime` element in the PL/Container configuration file, this is the value of the `id` element in the PL/Container configuration file. Maximum length is 63 Bytes.

You specify the name in the Greenplum Database UDF on the `# container` line.

{-s | --setting} *param=value*

Optional. Specify a setting to add to the runtime configuration information. You can specify this option multiple times. The setting applies to the runtime configuration specified by the *runtime_id*. The parameter is the XML attribute of the *`settings`* element in the PL/Container configuration file. These are valid parameters.

- `cpu_share` - Set the CPU limit for each container in the runtime configuration. The default value is 1024. The value is a relative weighting of CPU usage compared to other containers.
- `memory_mb` - Set the memory limit for each container in the runtime configuration. The default value is 1024. The value is an integer that specifies the amount of memory in MB.
- `resource_group_id` - Assign the specified resource group to the runtime configuration. The resource group limits the total CPU and memory resource usage for all containers that share this runtime configuration. You must specify the `groupid` of the resource group. For information about managing PL/Container resources, see *About PL/Container Resource Management*.

- `roles` - Specify the Greenplum Database roles that are allowed to run a container for the runtime configuration. You can specify a single role name or comma separated lists of role names. The default is no restriction.
- `use_container_logging` - Enable or disable Docker logging for the container. The value is either `yes` (enable logging) or `no` (disable logging, the default).

  The Greenplum Database server configuration parameter *`log_min_messages`* controls the log level. The default log level is `warning`. For information about PL/Container log information, see *Notes*.

{-v | --volume} ***shared-volume***

Optional. Specify a Docker volume to bind mount. You can specify this option multiple times to define multiple volumes.

The format for a shared volume: *`host-dir`*`:`*`container-dir`*`:[rw|ro]`. The information is stored as attributes in the `shared_directory` element of the `runtime` element in the PL/Container configuration file.

- *host-dir* - absolute path to a directory on the host system. The Greenplum Database administrator user (gpadmin) must have appropriate access to the directory.
- *container-dir* - absolute path to a directory in the Docker container.
- `[rw|ro]` - read-write or read-only access to the host directory from the container.

When adding configuration information for a new runtime, the utility adds this read-only shared volume information.

```
greenplum-home/bin/plcontainer_clients:/clientdir:ro
```

If needed, you can specify other shared directories. The utility returns an error if the specified *container-dir* is the same as the one that is added by the utility, or if you specify multiple shared volumes with the same *container-dir*.

Warning: Allowing read-write access to a host directory requires special considerations.

- When specifying read-write access to host directory, ensure that the specified host directory has the correct permissions.
- When running PL/Container user-defined functions, multiple concurrent Docker containers that are running on a host could change data in the host directory. Ensure that the functions support multiple concurrent access to the data in the host directory.

runtime-backup {-f | --file} ***config_file***

Copies the PL/Container configuration file to the specified file on the local host.

runtime-delete {-r | --runtime} ***runtime_id***

Removes runtime configuration information in the PL/Container configuration file on all Greenplum Database instances. The utility returns a message if the specified *runtime_id* does not exist in the file.

runtime-edit [{-e | --editor} ***editor*****]**

Edit the XML file `plcontainer_configuration.xml` with the specified editor. The default editor is `vi`.

Saving the file updates the configuration file on all Greenplum Database hosts. If errors exist in the updated file, the utility returns an error and does not update the file.

runtime-replace ***options***

Replaces runtime configuration information in the PL/Container configuration file on all Greenplum Database instances. If the *runtime_id* does not exist, the information is added

to the configuration file. The utility adds a startup command and shared directory to the configuration.

See *runtime-add* for command options and information added to the configuration.

runtime-restore {-f | --file} *config_file*

Replaces information in the PL/Container configuration file `plcontainer_configuration.xml` on all Greenplum Database instances with the information from the specified file on the local host.

runtime-show [{-r | --runtime} *runtime_id*]

Displays formatted PL/Container runtime configuration information. If a *runtime_id* is not specified, the configuration for all runtime IDs are displayed.

runtime-verify

Checks the PL/Container configuration information on the Greenplum Database instances with the configuration information on the master. If the utility finds inconsistencies, you are prompted to replace the remote copy with the local copy. The utility also performs XML validation.

-h | --help

Display help text. If specified without a command, displays help for all `plcontainer` commands. If specified with a command, displays help for the command.

--verbose

Enable verbose logging for the command.

Examples

These are examples of common commands to manage PL/Container:

- Install a Docker image on all Greenplum Database hosts. This example loads a Docker image from a file. The utility displays progress information on the command line as the utility installs the Docker image on all the hosts.

  ```
  plcontainer image-add -f plc_newr.tar.gz
  ```

  After installing the Docker image, you add or update a runtime entry in the PL/Container configuration file to give PL/Container access to the Docker image to start Docker containers.
- Install the Docker image only on the local Greenplum Database host:

  ```
  plcontainer image-add -f /home/gpadmin/plc_python_image.tar.gz --
  use_local_copy
  ```
- Add a container entry to the PL/Container configuration file. This example adds configuration information for a PL/R runtime, and specifies a shared volume and settings for memory and logging.

  ```
  plcontainer runtime-add -r runtime2 -i test_image2:0.1 -l r \
    -v /host_dir2/shared2:/container_dir2/shared2:ro \
    -s memory_mb=512 -s use_container_logging=yes
  ```

  The utility displays progress information on the command line as it adds the runtime configuration to the configuration file and distributes the updated configuration to all instances.
- Show specific runtime with given runtime id in configuration file

  ```
  plcontainer runtime-show -r plc_python_shared
  ```

The utility displays the configuration information similar to this output.

```
PL/Container Runtime Configuration:
------------------------------------------------------------
 Runtime ID: plc_python_shared
 Linked Docker Image: test1:latest
 Runtime Setting(s):
 Shared Directory:
 ---- Shared Directory From HOST '/usr/local/greenplum-db/bin/
plcontainer_clients' to Container '/clientdir', access mode is 'ro'
 ---- Shared Directory From HOST '/home/gpadmin/share/' to Container '/
opt/share', access mode is 'rw'
------------------------------------------------------------
```

- Edit the configuration in an interactive editor of your choice. This example edits the configuration file with the vim editor.

  ```
  plcontainer runtime-edit -e vim
  ```

  When you save the file, the utility displays progress information on the command line as it distributes the file to the Greenplum Database hosts.
- Save the current PL/Container configuration to a file. This example saves the file to the local file `/home/gpadmin/saved_plc_config.xml`

  ```
  plcontainer runtime-backup -f /home/gpadmin/saved_plc_config.xml
  ```
- Overwrite PL/Container configuration file with an XML file. This example replaces the information in the configuration file with the information from the file in the `/home/gpadmin` directory.

  ```
  plcontainer runtime-restore -f /home/gpadmin/
  new_plcontainer_configuration.xml
  ```

  The utility displays progress information on the command line as it distributes the updated file to the Greenplum Database instances.

plcontainer Configuration File

The Greenplum Database utility `plcontainer` manages the PL/Container configuration files in a Greenplum Database system. The utility ensures that the configuration files are consistent across the Greenplum Database master and segment instances.

> **Warning:** Modifying the configuration files on the segment instances without using the utility might create different, incompatible configurations on different Greenplum Database segments that could cause unexpected behavior.

PL/Container Configuration File

PL/Container maintains a configuration file `plcontainer_configuration.xml` in the data directory of all Greenplum Database segments. This query lists the Greenplum Database system data directories:

```
SELECT hostname, datadir FROM gp_segment_configuration;
```

A sample PL/Container configuration file is in `$GPHOME/share/postgresql/plcontainer`.

In an XML file, names, such as element and attribute names, and values are case sensitive.

In this XML file, the root element `configuration` contains one or more `runtime` elements. You specify the `id` of the `runtime` element in the `# container:` line of a PL/Container function definition.

This is an example file. Note that all XML elements, names, and attributes are case sensitive.

```
<?xml version="1.0" ?>
<configuration>
    <runtime>
        <id>plc_python_example1</id>
        <image>pivotaldata/plcontainer_python_with_clients:0.1</image>
        <command>./pyclient</command>
    </runtime>
    <runtime>
        <id>plc_python_example2</id>
        <image>pivotaldata/plcontainer_python_without_clients:0.1</image>
        <command>/clientdir/pyclient.sh</command>
        <shared_directory access="ro" container="/clientdir" host="/usr/
local/greenplum-db/bin/plcontainer_clients"/>
        <setting memory_mb="512"/>
        <setting use_container_logging="yes"/>
        <setting cpu_share="1024"/>
        <setting resource_group_id="16391"/>
    </runtime>
    <runtime>
        <id>plc_r_example</id>
        <image>pivotaldata/plcontainer_r_without_clients:0.2</image>
        <command>/clientdir/rclient.sh</command>
        <shared_directory access="ro" container="/clientdir" host="/usr/
local/greenplum-db/bin/plcontainer_clients"/>
        <setting use_container_logging="yes"/>
        <setting roles="gpadmin,user1"/>
    </runtime>
    <runtime>
</configuration>
```

These are the XML elements and attributes in a PL/Container configuration file.

`configuration`

Root element for the XML file.

`runtime`

One element for each specific container available in the system. These are child elements of the `configuration` element.

`id`

Required. The value is used to reference a Docker container from a PL/Container user-defined function. The `id` value must be unique in the configuration. The `id` must start with a character or digit (a-z, A-Z, or 0-9) and can contain characters, digits, or the characters _ (underscore), `.` (period), or `-` (dash). Maximum length is 63 Bytes.

The `id` specifies which Docker image to use when PL/Container creates a Docker container to execute a user-defined function.

`image`

Required. The value is the full Docker image name, including image tag. The same way you specify them for starting this container in Docker. Configuration allows to have many container objects referencing the same image name, this way in Docker they would be represented by identical containers.

For example, you might have two `runtime` elements, with different `id` elements, `plc_python_128` and `plc_python_256`, both referencing the Docker image `pivotaldata/plcontainer_python:1.0.0`. The first `runtime` specifies a 128MB RAM limit and the second one specifies a 256MB limit that is specified by the `memory_mb` attribute of a `setting` element.

command

Required. The value is the command to be run inside of container to start the client process inside in the container. When creating a `runtime` element, the `plcontainer` utility adds a `command` element based on the language (the `-l` option).

`command` element for the Python 2 language.

```
<command>/clientdir/pyclient.sh</command>
```

`command` element for the Python 3 language.

```
<command>/clientdir/pyclient3.sh</command>
```

`command` element for the R language.

```
<command>/clientdir/rclient.sh</command>
```

You should modify the value only if you build a custom container and want to implement some additional initialization logic before the container starts.

> **Note:** This element cannot be set with the `plcontainer` utility. You can update the configuration file with the `plcontainer runtime-edit` command.

shared_directory

Optional. This element specifies a shared Docker shared volume for a container with access information. Multiple `shared_directory` elements are allowed. Each `shared_directory` element specifies a single shared volume. XML attributes for the `shared_directory` element:

- `host` - a directory location on the host system.
- `container` - a directory location inside of container.
- `access` - access level to the host directory, which can be either `ro` (read-only) or `rw` (read-write).

When creating a `runtime` element, the `plcontainer` utility adds a `shared_directory` element.

```
<shared_directory access="ro" container="/clientdir" host="/usr/
local/greenplum-db/bin/plcontainer_clients"/>
```

For each `runtime` element, the `container` attribute of the `shared_directory` elements must be unique. For example, a `runtime` element cannot have two `shared_directory` elements with attribute `container="/clientdir"`.

> **Warning:** Allowing read-write access to a host directory requires special consideration.
>
> - When specifying read-write access to host directory, ensure that the specified host directory has the correct permissions.
> - When running PL/Container user-defined functions, multiple concurrent Docker containers that are running on a host could change data in the host directory. Ensure that the functions support multiple concurrent access to the data in the host directory.

settings

Optional. This element specifies Docker container configuration information. Each `setting` element contains one attribute. The element attribute specifies logging, memory, or networking information. For example, this element enables logging.

```
<setting use_container_logging="yes"/>
```

These are the valid attributes.

`cpu_share`

Optional. Specify the CPU usage for each PL/Container container in the runtime. The value of the element is a positive integer. The default value is 1024. The value is a relative weighting of CPU usage compared to other containers.

For example, a container with a `cpu_share` of 2048 is allocated double the CPU slice time compared with container with the default value of 1024.

`memory_mb="`*`size`*`"`

Optional. The value specifies the amount of memory, in MB, that each container is allowed to use. Each container starts with this amount of RAM and twice the amount of swap space. The container memory consumption is limited by the host system `cgroups` configuration, which means in case of memory overcommit, the container is terminated by the system.

`resource_group_id="`*`rg_groupid`*`"`

Optional. The value specifies the `groupid` of the resource group to assign to the PL/Container runtime. The resource group limits the total CPU and memory resource usage for all running containers that share this runtime configuration. You must specify the `groupid` of the resource group. If you do not assign a resource group to a PL/Container runtime configuration, its container instances are limited only by system resources. For information about managing PL/Container resources, see *About PL/Container Resource Management*.

`roles="`*`list_of_roles`*`"`

Optional. The value is a Greenplum Database role name or a comma-separated list of roles. PL/Container runs a container that uses the PL/Container runtime configuration only for the listed roles. If the attribute is not specified, any Greenplum Database role can run an instance of this container runtime configuration. For example, you create a UDF that specifies the `plcontainer` language and identifies a `# container:` runtime configuration that has the `roles` attribute set. When a role (user) runs the UDF, PL/Container checks the list of roles and runs the container only if the role is on the list.

`use_container_logging="{yes | no}"`

Optional. Enables or disables Docker logging for the container. The attribute value `yes` enables logging. The attribute value `no` disables logging (the default).

The Greenplum Database server configuration parameter *`log_min_messages`* controls the PL/Container log level. The default log level is `warning`. For information about PL/Container log information, see *Notes*.

By default, the PL/Container log information is sent to a system service. On Red Hat 7 or CentOS 7 systems, the log information is sent to the `journald` service. On Red Hat 6 or CentOS 6 systems, the log is sent to the `syslogd` service.

Update the PL/Container Configuration

You can add a `runtime` element to the PL/Container configuration file with the `plcontainer runtime-add` command. The command options specify information such as the runtime ID, Docker image, and language. You can use the `plcontainer runtime-replace` command to update an existing `runtime` element. The utility updates the configuration file on the master and all segment instances.

The PL/Container configuration file can contain multiple `runtime` elements that reference the same Docker image specified by the XML element `image`. In the example configuration file, the `runtime` elements contain `id` elements named `plc_python_128` and `plc_python_256`, both referencing the Docker container `pivotaldata/plcontainer_python:1.0.0`. The first `runtime` element is defined with a 128MB RAM limit and the second one with a 256MB RAM limit.

```
<configuration>
  <runtime>
    <id>plc_python_128</id>
    <image>pivotaldata/plcontainer_python:1.0.0</image>
    <command>./client</command>
    <shared_directory access="ro" container="/clientdir" host="/usr/local/
gpdb/bin/plcontainer_clients"/>
    <setting memory_mb="128"/>
  </runtime>
  <runtime>
    <id>plc_python_256</id>
    <image>pivotaldata/plcontainer_python:1.0.0</image>
    <command>./client</command>
    <shared_directory access="ro" container="/clientdir" host="/usr/local/
gpdb/bin/plcontainer_clients"/>
    <setting memory_mb="256"/>
    <setting resource_group_id="16391"/>
  </runtime>
<configuration>
```

Configuration changes that are made with the utility are applied to the XML files on all Greenplum Database segments. However, PL/Container configurations of currently running sessions use the configuration that existed during session start up. To update the PL/Container configuration in a running session, execute this command in the session.

```
SELECT * FROM plcontainer_refresh_config;
```

Running the command executes a PL/Container function that updates the session configuration on the master and segment instances.

psql

Interactive command-line interface for Greenplum Database

Synopsis

```
psql [option ...] [dbname [username]]
```

Description

`psql` is a terminal-based front-end to Greenplum Database. It enables you to type in queries interactively, issue them to Greenplum Database, and see the query results. Alternatively, input can be from a file. In addition, it provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

Options

`-a | --echo-all`

Print all nonempty input lines to standard output as they are read. (This does not apply to lines read interactively.) This is equivalent to setting the variable *ECHO* to `all`.

`-A | --no-align`

Switches to unaligned output mode. (The default output mode is aligned.)

-c '*command*' | --command='*command*'

Specifies that `psql` is to execute the specified command string, and then exit. This is useful in shell scripts. *command* must be either a command string that is completely parseable by the server, or a single backslash command. Thus you cannot mix SQL and `psql` meta-commands with this option. To achieve that, you could pipe the string into `psql`, like this:

```
echo '\x \\ SELECT * FROM foo;' | psql
```

(`\\` is the separator meta-command.)

If the command string contains multiple SQL commands, they are processed in a single transaction, unless there are explicit `BEGIN/COMMIT` commands included in the string to divide it into multiple transactions. This is different from the behavior when the same string is fed to `psql`'s standard input. Also, only the result of the last SQL command is returned.

-d *dbname* | --dbname=*dbname*

Specifies the name of the database to connect to. This is equivalent to specifying dbname as the first non-option argument on the command line.

If this parameter contains an = sign or starts with a valid URI prefix (`postgresql://` or `postgres://`), it is treated as a `conninfo` string. See *Connection Strings* in the PostgreSQL documentation for more information.

-e | --echo-queries

Copy all SQL commands sent to the server to standard output as well.

-E | --echo-hidden

Echo the actual queries generated by `\d` and other backslash commands. You can use this to study `psql`'s internal operations. This is equivalent to setting the variable *ECHO_HIDDEN* to `on`.

-f *filename* | --file=*filename*

Use the file *filename* as the source of commands instead of reading commands interactively. After the file is processed, `psql` terminates. This is in many ways equivalent to the meta-command `\i`.

If filename is `-` (hyphen), then standard input is read until an EOF indication or `\q` meta-command. Note however that Readline is not used in this case (much as if `-n` had been specified).

Using this option is subtly different from writing `psql < *filename*`. In general, both will do what you expect, but using `-f` enables some nice features such as error messages with line numbers. There is also a slight chance that using this option will reduce the start-up overhead. On the other hand, the variant using the shell's input redirection is (in theory) guaranteed to yield exactly the same output you would have received had you entered everything by hand.

-F *separator* | --field-separator=*separator*

Use the specified separator as the field separator for unaligned output.

-H | --html

Turn on HTML tabular output.

-l | --list

List all available databases, then exit. Other non-connection options are ignored.

-L *filename* | --log-file=*filename*

Write all query output into the specified log file, in addition to the normal output destination.

-n | --no-readline

Do not use Readline for line editing and do not use the command history. This can be useful to turn off tab expansion when cutting and pasting.

-o *filename* | --output=*filename*

Put all query output into the specified file.

-P *assignment* | --pset=*assignment*

Allows you to specify printing options in the style of `\pset` on the command line. Note that here you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write `-P format=latex`.

-q | --quiet

Specifies that `psql` should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option. This is equivalent to setting the variable *QUIET* to `on`.

-R *separator* | --record-separator=*separator*

Use *separator* as the record separator for unaligned output.

-s | --single-step

Run in single-step mode. That means the user is prompted before each command is sent to the server, with the option to cancel execution as well. Use this to debug scripts.

-S | --single-line

Runs in single-line mode where a new line terminates an SQL command, as a semicolon does.

-t | --tuples-only

Turn off printing of column names and result row count footers, etc. This command is equivalent to `\pset tuples_only` and is provided for convenience.

-T *table_options* | --table-attr= *table_options*

Allows you to specify options to be placed within the HTML table tag. See `\pset` for details.

-v *assignment* | --set=*assignment* | --variable= *assignment*

Perform a variable assignment, like the `\set` meta command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. To set a variable with an empty value, use the equal sign but leave off the value. These assignments are done during a very early stage of start-up, so variables reserved for internal purposes might get overwritten later.

-V | --version

Print the `psql` version and exit.

-x | --expanded

Turn on the expanded table formatting mode.

-X | --no-psqlrc

Do not read the start-up file (neither the system-wide `psqlrc` file nor the user's `~/.psqlrc` file).

-z | --field-separator-zero

Set the field separator for unaligned output to a zero byte.

-0 | --record-separator-zero

Set the record separator for unaligned output to a zero byte. This is useful for interfacing, for example, with `xargs -0`.

-1 | --single-transaction

When `psql` executes a script, adding this option wraps `BEGIN`/`COMMIT` around the script to execute it as a single transaction. This ensures that either all the commands complete successfully, or no changes are applied.

If the script itself uses `BEGIN`, `COMMIT`, or `ROLLBACK`, this option will not have the desired effects. Also, if the script contains any command that cannot be executed inside a transaction block, specifying this option will cause that command (and hence the whole transaction) to fail.

`-? | --help`

Show help about `psql` command line arguments, and exit.

Connection Options

`-h *host* | --host=*host*`

The host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

When starting `psql` on the master host, if the *host* value begins with a slash, it is used as the directory for the UNIX-domain socket.

`-p *port* | --port=*port*`

The TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

`-U *username* | --username=*username*`

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

`-W | --password`

Force a password prompt. `psql` should automatically prompt for a password whenever the server requests password authentication. However, currently password request detection is not totally reliable, hence this option to force a prompt. If no password prompt is issued and the server requires password authentication, the connection attempt will fail.

`-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a .pgpass file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

Note: This option remains set for the entire session, and so it affects uses of the meta-command `\connect` as well as the initial connection attempt.

Exit Status

`psql` returns 0 to the shell if it finished normally, 1 if a fatal error of its own (out of memory, file not found) occurs, 2 if the connection to the server went bad and the session was not interactive, and 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set.

Usage

Connecting to a Database

`psql` is a client application for Greenplum Database. In order to connect to a database you need to know the name of your target database, the host name and port number of the Greenplum master server and what database user name you want to connect as. `psql` can be told about those parameters via command line options, namely `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option it will be interpreted as the database name (or the user name, if the database name is already given). Not all of these options are required; there are useful defaults. If you omit the host name, `psql` will connect via a UNIX-domain socket to a master server on the local host, or via TCP/IP to `localhost` on

machines that do not have UNIX-domain sockets. The default master port number is 5432. If you use a different port for the master, you must specify the port. The default database user name is your operating-system user name, as is the default database name. Note that you cannot just connect to any database under any user name. Your database administrator should have informed you about your access rights.

When the defaults are not right, you can save yourself some typing by setting any or all of the environment variables `PGAPPNAME`, `PGDATABASE`, `PGHOST`, `PGPORT`, and `PGUSER` to appropriate values.

It is also convenient to have a `~/.pgpass` file to avoid regularly having to type in passwords. This file should reside in your home directory and contain lines of the following format:

```
hostname:port:database:username:password
```

The permissions on `.pgpass` must disallow any access to world or group (for example: `chmod 0600 ~/.pgpass`). If the permissions are less strict than this, the file will be ignored. (The file permissions are not currently checked on Microsoft Windows clients, however.)

An alternative way to specify connection parameters is in a `conninfo` string or a URI, which is used instead of a database name. This mechanism gives you very wide control over the connection. For example:

```
$ psql "service=myservice sslmode=require"
$ psql postgresql://gpmaster:5433/mydb?sslmode=require
```

This way you can also use LDAP for connection parameter lookup as described in *LDAP Lookup of Connection Parameters* in the PostgreSQL documentation. See *Parameter Keywords* in the PostgreSQL documentation for more information on all the available connection options.

If the connection could not be made for any reason (insufficient privileges, server is not running, etc.), `psql` will return an error and terminate.

If at least one of standard input or standard output are a terminal, then `psql` sets the client encoding to `auto`, which will detect the appropriate client encoding from the locale settings (`LC_CTYPE` environment variable on Unix systems). If this doesn't work out as expected, the client encoding can be overridden using the environment variable `PGCLIENTENCODING`.

Entering SQL Commands

In normal operation, `psql` provides a prompt with the name of the database to which `psql` is currently connected, followed by the string **=>** for a regular user or **=#** for a superuser. For example:

```
testdb=>
testdb=#
```

At the prompt, the user may type in SQL commands. Ordinarily, input lines are sent to the server when a command-terminating semicolon is reached. An end of line does not terminate a command. Thus commands can be spread over several lines for clarity. If the command was sent and executed without error, the results of the command are displayed on the screen.

If untrusted users have access to a database that has not adopted a *secure schema usage pattern*, begin your session by removing publicly-writable schemas from *search_path*. You can add `options=-csearch_path=` to the connection string or issue `SELECT pg_catalog.set_config('search_path', '', false)` before other SQL commands. This consideration is not specific to `psql`; it applies to every interface for executing arbitrary SQL commands.

Meta-Commands

Anything you enter in `psql` that begins with an unquoted backslash is a `psql` meta-command that is processed by `psql` itself. These commands help make `psql` more useful for administration or scripting. Meta-commands are more commonly called slash or backslash commands.

The format of a `psql` command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of whitespace characters.

To include whitespace into an argument you may quote it with single quotes. To include a single quote into such an argument, write two single quotes within single-quoted text. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\b` (backspace), `\r` (carriage return), `\f` (form feed), `\digits` (octal), and `\xdigits` (hexadecimal). A backslash preceding any other character within single-quoted text quotes that single character, whatever it is.

Within an argument, text that is enclosed in backquotes (`` ` ``) is taken as a command line that is passed to the shell. The output of the command (with any trailing newline removed) replaces the backquoted text.

If an unquoted colon (`:`) followed by a `psql` variable name appears within an argument, it is replaced by the variable's value, as described in *SQL Interpolation*.

Some commands take an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL: Unquoted letters are forced to lowercase, while double quotes (`"`) protect letters from case conversion and allow incorporation of whitespace into the identifier. Within double quotes, paired double quotes reduce to a single double quote in the resulting name. For example, `FOO"BAR"BAZ` is interpreted as `fooBARbaz`, and `"A weird"" name"` becomes `A weird" name`.

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL commands, if any. That way SQL and `psql` commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

`\a`

If the current table output format is unaligned, it is switched to aligned. If it is not unaligned, it is set to unaligned. This command is kept for backwards compatibility. See `\pset` for a more general solution.

`\c | \connect [`*`dbname`*` [`*`username`*`] [`*`host`*`] [`*`port`*`]] | `*`conninfo`*

Establishes a new Greenplum Database connection. The connection parameters to use can be specified either using a positional syntax, or using `conninfo` connection strings as detailed in *libpq Connection Strings*.

Where the command omits database name, user, host, or port, the new connection can reuse values from the previous connection. By default, values from the previous connection are reused except when processing a `conninfo` string. Passing a first argument of `-reuse-previous=on` or `-reuse-previous=off` overrides that default. When the command neither specifies nor reuses a particular parameter, the `libpq` default is used. Specifying any of *dbname*, *username*, *host* or *port* as `-` is equivalent to omitting that parameter.

If the new connection is successfully made, the previous connection is closed. If the connection attempt failed, the previous connection will only be kept if `psql` is in interactive mode. When executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos, and a safety mechanism that scripts are not accidentally acting on the wrong database.

Examples:

```
=> \c mydb myuser host.dom 6432
=> \c service=foo
=> \c "host=localhost port=5432 dbname=mydb connect_timeout=10
 sslmode=disable"
=> \c postgresql://tom@localhost/mydb?application_name=myapp
```

`\C [`*`title`*`]`

Sets the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to `\pset title`.

`\cd [`*`directory`*`]`

Changes the current working directory. Without argument, changes to the current user's home directory. To print your current working directory, use `\!pwd`.

`\conninfo`

Displays information about the current connection including the database name, the user name, the type of connection (UNIX domain socket, `TCP/IP`, etc.), the host, and the port.

`\copy {`*`table`*` [(`*`column_list`*`)] | (`*`query`*`)} {from | to} {'`*`filename`*`' | program '`*`command`*`' | stdin | stdout | pstdin | pstdout} [with] (`*`option`*` [, ...]) ]`

Performs a frontend (client) copy. This is an operation that runs an SQL *`COPY`* command, but instead of the server reading or writing the specified file, `psql` reads or writes the file and routes the data between the server and the local file system. This means that file accessibility and privileges are those of the local user, not the server, and no SQL superuser privileges are required.

When `program` is specified, *command* is executed by `psql` and the data from or to *command* is routed between the server and the client. This means that the execution privileges are those of the local user, not the server, and no SQL superuser privileges are required.

`\copy ... from stdin | to stdout` reads/writes based on the command input and output respectively. All rows are read from the same source that issued the command, continuing until `\.` is read or the stream reaches `EOF`. Output is sent to the same place as command output. To read/write from `psql`'s standard input or output, use `pstdin` or `pstdout`. This option is useful for populating tables in-line within a SQL script file.

The syntax of the command is similar to that of the SQL *`COPY`* command, and *option* must indicate one of the options of the SQL `COPY` command. Note that, because of this, special parsing rules apply to the `\copy` command. In particular, the variable substitution rules and backslash escapes do not apply.

This operation is not as efficient as the SQL `COPY` command because all data must pass through the client/server connection.

`\copyright`

Shows the copyright and distribution terms of PostgreSQL on which Greenplum Database is based.

`\d [`*`relation_pattern`*`] | \d+ [`*`relation_pattern`*`] | \dS [`*`relation_pattern`*`]`

For each relation (table, external table, view, materialized view, index, sequence, or foreign table) or composite type matching the relation pattern, show all columns, their types, the tablespace (if not the default) and any special attributes such as `NOT NULL` or defaults. Associated indexes, constraints, rules, and triggers are also shown. For foreign tables, the associated foreign server is shown as well.

- For some types of relation, `\d` shows additional information for each column: column values for sequences, indexed expressions for indexes, and foreign data wrapper options for foreign tables.
- The command form `\d+` is identical, except that more information is displayed: any comments associated with the columns of the table are shown, as is the presence of OIDs in the table, the view definition if the relation is a view.

  For partitioned tables, the command `\d` or `\d+` specified with the root partition table or child partition table displays information about the table including partition keys on the current level of the partition table. The command `\d+` also displays the immediate child partitions of the table and whether the child partition is an external table or regular table.

For append-optimized tables and column-oriented tables, `\d+` displays the storage options for a table. For append-optimized tables, the options are displayed for the table. For column-oriented tables, storage options are displayed for each column.

- By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

  Note: If `\d` is used without a pattern argument, it is equivalent to `\dtvmsE` which will show a list of all visible tables, views, materialized views, sequences, and foreign tables.

`\da[S] [`*`aggregate_pattern`*`]`

Lists aggregate functions, together with the data types they operate on. If a pattern is specified, only aggregates whose names match the pattern are shown. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

`\db[+] [`*`tablespace_pattern`*`]`

Lists all available tablespaces and their corresponding paths. If pattern is specified, only tablespaces whose names match the pattern are shown. If + is appended to the command name, each object is listed with its associated permissions.

`\dc[S+] [`*`conversion_pattern`*`]`

Lists conversions between character-set encodings. If a pattern is specified, only conversions whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If + is appended to the command name, each object is listed with its associated description.

`\dC[+] [`*`pattern`*`]`

Lists type casts. If a pattern is specified, only casts whose source or target types match the pattern are listed. If + is appended to the command name, each object is listed with its associated description.

`\dd[S] [`*`pattern`*`]`

Shows the descriptions of objects of type `constraint`, `operator class`, `operator family`, `rule`, and `trigger`. All other comments may be viewed by the respective backslash commands for those object types.

`\dd` displays descriptions for objects matching the pattern, or of visible objects of the appropriate type if no argument is given. But in either case, only objects that have a description are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

Descriptions for objects can be created with the `COMMENT` SQL command.

`\ddp [`*`pattern`*`]`

Lists default access privilege settings. An entry is shown for each role (and schema, if applicable) for which the default privilege settings have been changed from the built-in defaults. If *pattern* is specified, only entries whose role name or schema name matches the pattern are listed.

The *ALTER DEFAULT PRIVILEGES* command is used to set default access privileges. The meaning of the privilege display is explained under *GRANT*.

`\dD[S+] [`*`domain_pattern`*`]`

Lists domains. If a pattern is specified, only domains whose names match the pattern are shown. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If + is appended to the command name, each object is listed with its associated permissions and description.

`\dEimstPv[S+] [`*`external_table`*` | `*`index`*` | `*`materialized_view`*` | `*`sequence`*` | `*`table`*` | `*`parent table`*` | `*`view`*`]`

This is not the actual command name: the letters `E`, `i`, `m`, `s`, `t`, `P`, and `v` stand for external table, index, materialized view, sequence, table, parent table, and view, respectively. You can specify any or all of these letters, in any order, to obtain a listing of objects of these types. For example, `\dit` lists indexes and tables. If `+` is appended to the command name, each object is listed with its physical size on disk and its associated description, if any. If a pattern is specified, only objects whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

`\des[+] [`*`foreign_server_pattern`*`]`

Lists foreign servers. If a pattern is specified, only those servers whose name matches the pattern are listed. If the form `\des+` is used, a full description of each server is shown, including the server's ACL, type, version, options, and description.

`\det[+] [`*`foreign_table_pattern`*`]`

Lists all foreign tables. If a pattern is specified, only entries whose table name or schema name matches the pattern are listed. If the form `\det+` is used, generic options and the foreign table description are also displayed.

`\deu[+] [`*`user_mapping_pattern`*`]`

Lists user mappings. If a pattern is specified, only those mappings whose user names match the pattern are listed. If the form `\deu+` is used, additional information about each mapping is shown.

> **Warning:** `\deu+` might also display the user name and password of the remote user, so care should be taken not to disclose them.

`\dew[+] [`*`foreign_data_wrapper_pattern`*`]`

Lists foreign-data wrappers. If a pattern is specified, only those foriegn-data wrappers whose name matches the pattern are listed. If the form `\dew+` is used, the ACL, options, and description of the foreign-data wrapper are also shown.

`\df[antwS+] [`*`function_pattern`*`]`

Lists functions, together with their arguments, return types, and function types, which are classified as "agg" (aggregate), "normal", "trigger", or "window". To display only functions of a specific type(s), add the corresponding letters `a`, `n`, `t`, or `w`, to the command. If a pattern is specified, only functions whose names match the pattern are shown. If the form `\df+` is used, additional information about each function, including security, volatility, language, source code, and description, is shown. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

`\dF[+] [`*`pattern`*`]`

Lists text search configurations. If a pattern is specified, only configurations whose names match the pattern are shown. If the form `\dF+` is used, a full description of each configuration is shown, including the underlying text search parser and the dictionary list for each parser token type.

`\dFd[+] [`*`pattern`*`]`

Lists text search dictionaries. If a pattern is specified, only dictionaries whose names match the pattern are shown. If the form `\dFd+` is used, additional information is shown about each selected dictionary, including the underlying text search template and the option values.

`\dFp[+] [`*`pattern`*`]`

Lists text search parsers. If a pattern is specified, only parsers whose names match the pattern are shown. If the form `\dFp+` is used, a full description of each parser is shown, including the underlying functions and the list of recognized token types.

`\dFt[+] [`*`pattern`*`]`

Lists text search templates. If a pattern is specified, only templates whose names match the pattern are shown. If the form `\dFt+` is used, additional information is shown about each template, including the underlying function names.

`\dg[+] [role_pattern]`

Lists database roles. (Since the concepts of "users" and "groups" have been unified into "roles", this command is now equivalent to `\du`.) If a pattern is specified, only those roles whose names match the pattern are listed. If the form `\dg+` is used, additional information is shown about each role; currently this adds the comment for each role.

`\dl`

This is an alias for `\lo_list`, which shows a list of large objects.

> **Note:** Greenplum Database does not support the PostgreSQL *large object facility* for streaming user data that is stored in large-object structures.

`\dL[S+] [pattern]`

Lists procedural languages. If a pattern is specified, only languages whose names match the pattern are listed. By default, only user-created languages are shown; supply the `S` modifier to include system objects. If + is appended to the command name, each language is listed with its call handler, validator, access privileges, and whether it is a system object.

`\dn[S+] [schema_pattern]`

Lists all available schemas (namespaces). If a pattern is specified, only schemas whose names match the pattern are listed. By default, only user- create objects are show; supply a pattern or the `S` modifier to include system objects. If + is appended to the command name, each object is listed with its associated permissions and description, if any.

`\do[S] [operator_pattern]`

Lists available operators with their operand and return types. If a pattern is specified, only operators whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

`\dO[S+] [pattern]`

Lists collations. If a pattern is specified, only collations whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If + is appended to the command name, each collation is listed with its associated description, if any. Note that only collations usable with the current database's encoding are shown, so the results may vary in different databases of the same installation.

`\dp [relation_pattern_to_show_privileges]`

Lists tables, views, and sequences with their associated access privileges. If a pattern is specified, only tables, views, and sequences whose names match the pattern are listed. The *GRANT* and *REVOKE* commands are used to set access privileges. The meaning of the privilege display is explained under *GRANT*.

`\drds [role-pattern [database-pattern]]`

Lists defined configuration settings. These settings can be role-specific, database-specific, or both. *role-pattern* and *database-pattern* are used to select specific roles and database to list, respectively. If omitted, or if * is specified, all settings are listed, including those not role-specific or database-specific, respectively.

The *ALTER ROLE* and *ALTER DATABASE* commands are used to define per-role and per-database role configuration settings.

`\dT[S+] [datatype_pattern]`

Lists data types. If a pattern is specified, only types whose names match the pattern are listed. If + is appended to the command name, each type is listed with its internal name and size, its allowed values if it is an `enum` type, and its associated permissions. By

default, only user-created objects are shown; supply a pattern or the S modifier to include system objects.

\du[+] [*role_pattern*]

Lists database roles. (Since the concepts of "users" and "groups" have been unified into "roles", this command is now equivalent to \dg.) If a pattern is specified, only those roles whose names match the pattern are listed. If the form \du+ is used, additional information is shown about each role; currently this adds the comment for each role.

\dx[+] [*extension_pattern*]

Lists installed extensions. If a pattern is specified, only those extensions whose names match the pattern are listed. If the form \dx+ is used, all of the objects belonging to each matching extension are listed.

\dy[+] [*pattern*]

Lists event triggers. If a pattern is specified, only those triggers whose names match the pattern are listed. If + is appended to the command name, each object is listed with its associated description.

\dy[+] [*pattern*]

Lists event triggers. If a pattern is specified, only those triggers whose names match the pattern are listed. If + is appended to the command name, each object is listed with its associated description.

> **Note:** Greenplum Database does not support user-defined triggers.

\e | \edit [*filename*] [*line_number*]

If *filename* is specified, the file is edited; after the editor exits, its content is copied back to the query buffer. If no *filename* is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

The new query buffer is then re-parsed according to the normal rules of psql, where the whole buffer is treated as a single line. (Thus you cannot make scripts this way. Use \i for that.) This means also that if the query ends with (or rather contains) a semicolon, it is immediately executed. In other cases it will merely wait in the query buffer; type semicolon or \g to send it, or \r to cancel.

If a line number is specified, psql will position the cursor on the specified line of the file or query buffer. Note that if a single all-digits argument is given, psql assumes it is a line number, not a file name.

See *Environment* for information about configuring and customizing your editor.

\echo *text* [...]

Prints the arguments to the standard output, separated by one space and followed by a newline. This can be useful to intersperse information in the output of scripts. If the first argument is an unquoted -n, the trailing newline is not written.

> **Note:** If you use the \o command to redirect your query output you might wish to use \qecho instead of this command.

\ef [*function_description* [*line_number*]]

This command fetches and edits the definition of the named function, in the form of a CREATE OR REPLACE FUNCTION command. Editing is done in the same way as for \edit. After the editor exits, the updated command waits in the query buffer; type semicolon or \g to send it, or \r to cancel.

The target function can be specified by name alone, or by name and arguments, for example foo(integer, text). The argument types must be given if there is more than one function with the same name.

If no function is specified, a blank CREATE FUNCTION template is presented for editing.

If a line number is specified, `psql` will position the cursor on the specified line of the function body. (Note that the function body typically does not begin on the first line of the file.)

See *Environment* for information about configuring and customizing your editor.

`\encoding [`*`encoding`*`]`

Sets the client character set encoding. Without an argument, this command shows the current encoding.

`\f [`*`field_separator_string`*`]`

Sets the field separator for unaligned query output. The default is the vertical bar (`|`). See also `\pset` for a generic way of setting output options.

`\g [`*`filename`*`]`
`\g [ | `*`command`*` ]`

Sends the current query input buffer to the server, and optionally stores the query's output in *filename* or pipes the output to the shell command *command*. The file or command is written to only if the query successfully returns zero or more tuples, not if the query fails or is a non-data-returning SQL command.

A bare `\g` is essentially equivalent to a semi-colon. A `\g` with argument is a one-shot alternative to the `\o` command.

`\gset [`*`prefix`*`]`

Sends the current query input buffer to the server and stores the query's output into `psql` variables. The query to be executed must return exactly one row. Each column of the row is stored into a separate variable, named the same as the column. For example:

```
=> SELECT 'hello' AS var1, 10 AS var2;
-> \gset
=> \echo :var1 :var2
hello 10
```

If you specify a *prefix*, that string is prepended to the query's column names to create the variable names to use:

```
=> SELECT 'hello' AS var1, 10 AS var2;
-> \gset result_
=> \echo :result_var1 :result_var2
hello 10
```

If a column result is NULL, the corresponding variable is unset rather than being set.

If the query fails or does not return one row, no variables are changed.

`\h | \help [`*`sql_command`*`]`

Gives syntax help on the specified SQL command. If a command is not specified, then `psql` will list all the commands for which syntax help is available. If *command* is an asterisk (`*`) then syntax help on all SQL commands is shown. To simplify typing, commands that consist of several words do not have to be quoted.

`\H | \html`

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see `\pset` about setting other output options.

`\i | \include `*`filename`*

Reads input from the file *filename* and executes it as though it had been typed on the keyboard.

If *filename* is `-` (hyphen), then standard input is read until an EOF indication or `\q` meta-command. This can be used to intersperse interactive input with input from files. Note that Readline behavior will be used only if it is active at the outermost level.

If you want to see the lines on the screen as they are read you must set the variable `ECHO` to `all`.

`\ir | \include_relative` *`filename`*

The `\ir` command is similar to `\i`, but resolves relative file names differently. When executing in interactive mode, the two commands behave identically. However, when invoked from a script, `\ir` interprets file names relative to the directory in which the script is located, rather than the current working directory.

`\l[+] | \list[+]` [*`pattern`*]

List the databases in the server and show their names, owners, character set encodings, and access privileges. If a pattern is specified, only databases whose names match the pattern are listed. If + is appended to the command name, database sizes, default tablespaces, and descriptions are also displayed. (Size information is only available for databases that the current user can connect to.)

`\lo_export` *`loid filename`*

Reads the large object with OID *loid* from the database and writes it to *filename*. Note that this is subtly different from the server function `lo_export`, which acts with the permissions of the user that the database server runs as and on the server's file system. Use `\lo_list` to find out the large object's OID.

> **Note:** Greenplum Database does not support the PostgreSQL *large object facility* for streaming user data that is stored in large-object structures.

`\lo_import` *`large_object_filename`* [*`comment`*]

Stores the file into a large object. Optionally, it associates the given comment with the object. Example:

```
mydb=> \lo_import '/home/gpadmin/pictures/photo.xcf' 'a
picture of me'
lo_import 152801
```

The response indicates that the large object received object ID 152801 which one ought to remember if one wants to access the object ever again. For that reason it is recommended to always associate a human-readable comment with every object. Those can then be seen with the `\lo_list` command. Note that this command is subtly different from the server-side `lo_import` because it acts as the local user on the local file system, rather than the server's user and file system.

> **Note:** Greenplum Database does not support the PostgreSQL *large object facility* for streaming user data that is stored in large-object structures.

`\lo_list`

Shows a list of all large objects currently stored in the database, along with any comments provided for them.

> **Note:** Greenplum Database does not support the PostgreSQL *large object facility* for streaming user data that is stored in large-object structures.

`\lo_unlink` *`largeobject_oid`*

Deletes the large object of the specified OID from the database. Use `\lo_list` to find out the large object's OID.

> **Note:** Greenplum Database does not support the PostgreSQL *large object facility* for streaming user data that is stored in large-object structures.

\o | \out [*filename*]
\o | \out [| *command*]

Saves future query results to the file *filename* or pipes future results to the shell command *command*. If no argument is specified, the query output is reset to the standard output. Query results include all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as `\d`), but not error messages. To intersperse text output in between query results, use `\qecho`.

\p

Print the current query buffer to the standard output.

\password [*username*]

Changes the password of the specified user (by default, the current user). This command prompts for the new password, encrypts it, and sends it to the server as an `ALTER ROLE` command. This makes sure that the new password does not appear in cleartext in the command history, the server log, or elsewhere.

\prompt [*text*] *name*

Prompts the user to supply text, which is assigned to the variable *name*. An optional prompt string, *text*, can be specified. (For multiword prompts, surround the text with single quotes.)

By default, `\prompt` uses the terminal for input and output. However, if the `-f` command line switch was used, `\prompt` uses standard input and standard output.

\pset [*print_option* [*value*]]

This command sets options affecting the output of query result tables. *print_option* describes which option is to be set. The semantics of *value* vary depending on the selected option. For some options, omitting *value* causes the option to be toggled or unset, as described under the particular option. If no such behavior is mentioned, then omitting *value* just results in the current setting being displayed.

`\pset` without any arguments displays the current status of all printing options.

Adjustable printing options are:

- **`border`** – The *value* must be a number. In general, the higher the number the more borders and lines the tables will have, but this depends on the particular format. In HTML format, this will translate directly into the `border=...` attribute; in the other formats only values `0` (no border), `1` (internal dividing lines), and `2` (table frame) make sense. `latex` and `latex-longtable` also support a `border` value of 3 which adds a dividing line between each row.
- **`columns`** – Sets the target width for the `wrapped` format, and also the width limit for determining whether output is wide enough to require the pager or switch to the vertical display in expanded auto mode. The default is *zero*. Zero causes the target width to be controlled by the environment variable `COLUMNS`, or the detected screen width if `COLUMNS` is not set. In addition, if `columns` is zero then the wrapped format affects screen output only. If columns is nonzero then file and pipe output is wrapped to that width as well.

  After setting the target width, use the command `\pset format wrapped` to enable the wrapped format.
- **`expanded`** | **`x`** – If *value* is specified it must be either `on` or `off`, which will enable or disable expanded mode, or `auto`. If *value* is omitted the command toggles between the `on` and `off` settings. When expanded mode is enabled, query results are displayed in two columns, with the column name on the left and the data on the right. This mode is useful if the data wouldn't fit on the screen in the normal "horizontal" mode. In the `auto` setting, the expanded mode is used whenever the query output is wider than the screen, otherwise the regular mode is used. The `auto` setting is only effective in the

aligned and wrapped formats. In other formats, it always behaves as if the expanded mode is `off`.

- **`fieldsep`** – Specifies the field separator to be used in unaligned output mode. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep '\t'`. The default field separator is `'|'` (a vertical bar).
- **`fieldsep_zero`** - Sets the field separator to use in unaligned output format to a zero byte.
- **`footer`** – If *value* is specified it must be either `on` or `off` which will enable or disable display of the table footer (the (*n* rows) count). If *value* is omitted the command toggles footer display on or off.
- **`format`** – Sets the output format to one of `unaligned`, `aligned`, `html`, `latex` (uses `tabular`), `latex-longtable`, `troff-ms`, or `wrapped`. Unique abbreviations are allowed.

  `unaligned` format writes all columns of a row on one line, separated by the currently active field separator. This is useful for creating output that might be intended to be read in by other programs (for example, tab-separated or comma-separated format).

  `aligned` format is the standard, human-readable, nicely formatted text output; this is the default.

  The **`html`**, **`latex`**, **`latex-longtable`**, and **`troff-ms`** formats put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! (This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper. `latex-longtable` also requires the LaTeX `longtable` and `booktabs` packages.)

  The **`wrapped`** format is like `aligned`, but wraps wide data values across lines to make the output fit in the target column width. The target width is determined as described under the `columns` option. Note that `psql` does not attempt to wrap column header titles; the `wrapped` format behaves the same as `aligned` if the total width needed for column headers exceeds the target.
- **`linestyle`** [**`unicode`** | **`ascii`** | **`old-ascii`**] – Sets the border line drawing style to one of unicode, ascii, or old-ascii. Unique abbreviations, including one letter, are allowed for the three styles. The default setting is `ascii`. This option only affects the `aligned` and `wrapped` output formats.

  `ascii` – uses plain ASCII characters. Newlines in data are shown using a + symbol in the right-hand margin. When the wrapped format wraps data from one line to the next without a newline character, a dot (`.`) is shown in the right-hand margin of the first line, and again in the left-hand margin of the following line.

  `old-ascii` – style uses plain ASCII characters, using the formatting style used in PostgreSQL 8.4 and earlier. Newlines in data are shown using a `:` symbol in place of the left-hand column separator. When the data is wrapped from one line to the next without a newline character, a `;` symbol is used in place of the left-hand column separator.

  `unicode` – style uses Unicode box-drawing characters. Newlines in data are shown using a carriage return symbol in the right-hand margin. When the data is wrapped from one line to the next without a newline character, an ellipsis symbol is shown in the right-hand margin of the first line, and again in the left-hand margin of the following line.

  When the `border` setting is greater than zero, this option also determines the characters with which the border lines are drawn. Plain ASCII characters work everywhere, but Unicode characters look nicer on displays that recognize them.

- **`null 'string'`** – The second argument is a string to print whenever a column is null. The default is to print nothing, which can easily be mistaken for an empty string. For example, one might prefer `\pset null '(null)'`.
- **`numericlocale`** – If *value* is specified it must be either `on` or `off` which will enable or disable display of a locale-specific character to separate groups of digits to the left of the decimal marker. If *value* is omitted the command toggles between regular and locale-specific numberic output.
- **`pager`** – Controls the use of a pager for query and `psql` help output. If the environment variable `PAGER` is set, the output is piped to the specified program. Otherwise a platform-dependent default (such as `more`) is used. When `off`, the pager program is not used. When `on`, the pager is used only when appropriate, i.e. when the output is to a terminal and will not fit on the screen. Pager can also be set to `always`, which causes the pager to be used for all terminal output regardless of whether it fits on the screen. `\pset pager` without a *value* toggles pager use on and off.
- **`recordsep`** – Specifies the record (line) separator to use in unaligned output mode. The default is a newline character.
- **`recordsep_zero`** - Sets the record separator to use in unaligned output format to a zero byte.
- **`tableattr`** | **`T`** [*text*] – In HTML format, this specifies attributes to be placed inside the HTML `table` tag. This could for example be `cellpadding` or `bgcolor`. Note that you probably don't want to specify `border` here, as that is already taken care of by `\pset border`. If no *value* is given, the table attributes are unset.

  In `latex-longtable` format, this controls the proportional width of each column containing a left-aligned data type. It is specified as a whitespace-separated list of values, e.g. `'0.2 0.2 0.6'`. Unspecified output columns use the last specified value.
- **`title`** [*text*] – Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no *value* is given, the title is unset.
- **`tuples_only`** | **`t`** [*novalue* | *on* | *off*] – If *value* is specified, it must be either `on` or `off` which will enable or disable tuples-only mode. If *value* is omitted the command toggles between regular and tuples-only output. Regular output includes extra information such as column headers, titles, and various footers. In tuples-only mode, only actual table data is shown. The `\t` command is equivalent to `\psettuples_only` and is provided for convenience.

  Tip:

  There are various shortcut commands for `\pset`. See `\a`, `\C`, `\f`, `\H`, `\t`, `\T`, and `\x`.

`\q | \quit`

Quits the `psql` program. In a script file, only execution of that script is terminated.

`\qecho *text* [ ... ]`

This command is identical to `\echo` except that the output will be written to the query output channel, as set by `\o`.

`\r | \reset`

Resets (clears) the query buffer.

`\s [*filename*]`

Print `psql`'s command line history to `filename`. If `filename` is omitted, the history is written to the standard output (using the pager if appropriate). This command is not available if `psql` was built without `Readline` support.

`\set [*name* [*value* [ ... ]]]`

Sets the `psql` variable *name* to *value*, or if more than one value is given, to the concatenation of all of them. If only one argument is given, the variable is just set with an empty value. To unset a variable, use the `\unset` command.

`\set` without any arguments displays the names and values of all currently-set `psql` variables.

Valid variable names can contain characters, digits, and underscores. See "Variables" in *Advanced Features*. Variable names are case-sensitive.

Although you are welcome to set any variable to anything you want, `psql` treats several variables as special. They are documented in the topic about variables.

This command is unrelated to the SQL command *SET*.

`\setenv` *`name`* `[` *`value`* `]`

Sets the environment variable *name* to *value*, or if the *value* is not supplied, unsets the environment variable. Example:

```
testdb=> \setenv PAGER less
testdb=> \setenv LESS -imx4F
```

`\sf[+]` *`function_description`*

This command fetches and shows the definition of the named function, in the form of a `CREATE OR REPLACE FUNCTION` command. The definition is printed to the current query output channel, as set by `\o`.

The target function can be specified by name alone, or by name and arguments, for example `foo(integer, text)`. The argument types must be given if there is more than one function of the same name.

If + is appended to the command name, then the output lines are numbered, with the first line of the function body being line 1.

`\t [novalue | on | off]`

The `\t` command by itself toggles a display of output column name headings and row count footer. The values `on` and `off` set the tuples display, regardless of the current setting. This command is equivalent to `\pset tuples_only` and is provided for convenience.

`\T` *`table_options`*

Specifies attributes to be placed within the `table` tag in HTML output format. This command is equivalent to `\pset tableattr` *`table_options`*

`\timing [novalue | on | off]`

Without a parameter, toggles a display of how long each SQL statement takes, in milliseconds. The values `on` and `off` set the time display, regardless of the current setting.

`\unset` *`name`*

Unsets (deletes) the `psql` variable *name*.

`\w | \write` *`filename`*
`\w | \write |` *`command`*

Outputs the current query buffer to the file *filename* or pipes it to the shell command *command*.

`\watch [`*`seconds`*`]`

Repeatedly execute the current query buffer (like `\g`) until interrupted or the query fails. Wait the specified number of seconds (default 2) between executions.

`\x [ on | off | auto ]`

Sets or toggles expanded table formatting mode. As such it is equivalent to `\pset expanded`.

`\z [`*`pattern`*`]`

Lists tables, views, and sequences with their associated access privileges. If a pattern is specified, only tables, views and sequences whose names match the pattern are listed. This is an alias for `\dp`.

`\! [command]`

Escapes to a separate shell or executes the shell command *command*. The arguments are not further interpreted; the shell will see them as-is. In particular, the variable substitution rules and backslash escapes do not apply.

`\?`

Shows help information about the `psql` backslash commands.

Patterns

The various `\d` commands accept a pattern parameter to specify the object name(s) to be displayed. In the simplest case, a pattern is just the exact name of the object. The characters within a pattern are normally folded to lower case, just as in SQL names; for example, `\dt FOO` will display the table named `foo`. As in SQL names, placing double quotes around a pattern stops folding to lower case. Should you need to include an actual double quote character in a pattern, write it as a pair of double quotes within a double-quote sequence; again this is in accord with the rules for SQL quoted identifiers. For example, `\dt "FOO""BAR"` will display the table named `FOO"BAR` (not `foo"bar`). Unlike the normal rules for SQL names, you can put double quotes around just part of a pattern, for instance `\dt FOO"FOO"BAR` will display the table named `fooFOObar`.

Within a pattern, `*` matches any sequence of characters (including no characters) and `?` matches any single character. (This notation is comparable to UNIX shell file name patterns.) For example, `\dt int*` displays all tables whose names begin with `int`. But within double quotes, `*` and `?` lose these special meanings and are just matched literally.

A pattern that contains a dot (`.`) is interpreted as a schema name pattern followed by an object name pattern. For example, `\dt foo*.bar*` displays all tables whose table name starts with `bar` that are in schemas whose schema name starts with `foo`. When no dot appears, then the pattern matches only objects that are visible in the current schema search path. Again, a dot within double quotes loses its special meaning and is matched literally.

Advanced users can use regular-expression notations. All regular expression special characters work as specified in the *PostgreSQL documentation on regular expressions*, except for `.` which is taken as a separator as mentioned above, `*` which is translated to the regular-expression notation `.*`, and `?` which is translated to `.`. You can emulate these pattern characters at need by writing `?` for `.`, `(R+|)` for `R*`, or `(R|)` for `R?`. Remember that the pattern must match the whole name, unlike the usual interpretation of regular expressions; write `*` at the beginning and/or end if you don't wish the pattern to be anchored. Note that within double quotes, all regular expression special characters lose their special meanings and are matched literally. Also, the regular expression special characters are matched literally in operator name patterns (such as the argument of `\do`).

Whenever the pattern parameter is omitted completely, the `\d` commands display all objects that are visible in the current schema search path – this is equivalent to using the pattern `*.`. To see all objects in the database, use the pattern `*.*.`.

Advanced Features

Variables

`psql` provides variable substitution features similar to common UNIX command shells. Variables are simply name/value pairs, where the value can be any string of any length. The name must consist of letters (including non-Latin letters), digits, and underscores.

To set a variable, use the psql meta-command `\set`. For example,

```
testdb=> \set foo bar
```

sets the variable `foo` to the value `bar`. To retrieve the content of the variable, precede the name with a colon, for example:

```
testdb=> \echo :foo
bar
```

This works in both regular SQL commands and meta-commands; there is more detail in *SQL Interpolation*.

If you call `\set` without a second argument, the variable is set, with an empty string as *value*. To unset (i.e., delete) a variable, use the command `\unset`. To show the values of all variables, call `\set` without any argument.

> **Note:** The arguments of `\set` are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as `\set :foo 'something'` and get 'soft links' or 'variable variables' of Perl or PHP fame, respectively. Unfortunately, there is no way to do anything useful with these constructs. On the other hand, `\set bar :foo` is a perfectly valid way to copy a variable.

A number of these variables are treated specially by `psql`. They represent certain option settings that can be changed at run time by altering the value of the variable, or in some cases represent changeable state of `psql`. Although you can use these variables for other purposes, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables' names consist of all upper-case ASCII letters (and possibly digits and underscores). To ensure maximum compatibility in the future, avoid using such variable names for your own purposes. A list of all specially treated variables follows.

AUTOCOMMIT

When on (the default), each SQL command is automatically committed upon successful completion. To postpone commit in this mode, you must enter a `BEGIN` or `START TRANSACTION` SQL command. When off or unset, SQL commands are not committed until you explicitly issue `COMMIT` or `END`. The autocommit-on mode works by issuing an implicit `BEGIN` for you, just before any command that is not already in a transaction block and is not itself a `BEGIN` or other transaction-control command, nor a command that cannot be executed inside a transaction block (such as `VACUUM`).

In autocommit-off mode, you must explicitly abandon any failed transaction by entering `ABORT` or `ROLLBACK`. Also keep in mind that if you exit the session without committing, your work will be lost.

The autocommit-on mode is PostgreSQL's traditional behavior, but autocommit-off is closer to the SQL spec. If you prefer autocommit-off, you may wish to set it in your `~/.psqlrc` file.

COMP_KEYWORD_CASE

Determines which letter case to use when completing an SQL key word. If set to `lower` or `upper`, the completed word will be in lower or upper case, respectively. If set to `preserve-lower` or `preserve-upper` (the default), the completed word will be in the case of the word already entered, but words being completed without anything entered will be in lower or upper case, respectively.

DBNAME

The name of the database you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

ECHO

If set to `all`, all nonempty input lines are printed to standard output as they are read. (This does not apply to lines read interactively.) To select this behavior on program start-up, use the switch `-a`. If set to queries, `psql` prints each query to standard output as it is sent to the server. The switch for this is `-e`.

ECHO_HIDDEN

When this variable is set to on and a backslash command queries the database, the query is first shown. This feature helps you to study Greenplum Database internals and provide similar functionality in your own programs. (To select this behavior on program start-up, use the switch -E.) If you set the variable to the value noexec, the queries are just shown but are not actually sent to the server and executed.

ENCODING

The current client character set encoding.

FETCH_COUNT

If this variable is set to an integer value > 0, the results of SELECT queries are fetched and displayed in groups of that many rows, rather than the default behavior of collecting the entire result set before display. Therefore only a limited amount of memory is used, regardless of the size of the result set. Settings of 100 to 1000 are commonly used when enabling this feature. Keep in mind that when using this feature, a query may fail after having already displayed some rows.

Although you can use any output format with this feature, the default aligned format tends to look bad because each group of FETCH_COUNT rows will be formatted separately, leading to varying column widths across the row groups. The other output formats work better.

HISTCONTROL

If this variable is set to ignorespace, lines which begin with a space are not entered into the history list. If set to a value of ignoredups, lines matching the previous history line are not entered. A value of ignoreboth combines the two options. If unset, or if set to any other value than those above, all lines read in interactive mode are saved on the history list.

HISTFILE

The file name that will be used to store the history list. The default value is ~/.psql_history. For example, putting

```
\set HISTFILE ~/.psql_history- :DBNAME
```

in ~/.psqlrc will cause psql to maintain a separate history for each database.

HISTSIZE

The number of commands to store in the command history. The default value is 500.

HOST

The database server host you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

IGNOREEOF

If unset, sending an EOF character (usually CTRL+D) to an interactive session of psql will terminate the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

LASTOID

The value of the last affected OID, as returned from an INSERT or lo_import command. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

ON_ERROR_ROLLBACK

When set to on, if a statement in a transaction block generates an error, the error is ignored and the transaction continues. When set to interactive, such errors are only ignored in interactive sessions, and not when reading script files. When unset or set to off, a statement in a transaction block that generates an error aborts the entire

transaction. The error rollback mode works by issuing an implicit `SAVEPOINT` for you, just before each command that is in a transaction block, and rolls back to the savepoint on error.

`ON_ERROR_STOP`

By default, command processing continues after an error. When this variable is set to `on`, processing will instead stop immediately. In interactive mode, `psql` will return to the command prompt; otherwise, `psql` will exit, returning error code 3 to distinguish this case from fatal error conditions, which are reported using error code 1. In either case, any currently running scripts (the top-level script, if any, and any other scripts which it may have in invoked) will be terminated immediately. If the top-level command string contained multiple SQL commands, processing will stop with the current command.

`PORT`

The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be unset.

`PROMPT1`
`PROMPT2`
`PROMPT3`

These specify what the prompts `psql` issues should look like. See "Prompting".

`QUIET`

Setting this variable to `on` is equivalent to the command line option `-q`. It is not very useful in interactive mode.

`SINGLELINE`

This variable is equivalent to the command line option `-S`.

`SINGLESTEP`

Setting this variable to `on` is equivalent to the command line option `-s`.

`USER`

The database user you are currently connected as. This is set every time you connect to a database (including program start-up), but can be unset.

`VERBOSITY`

This variable can be set to the values `default`, `verbose`, or `terse` to control the verbosity of error reports.

SQL Interpolation

A key feature of `psql` variables is that you can substitute ("interpolate") them into regular SQL statements, as well as the arguments of meta-commands. Furthermore, `psql` provides facilities for ensuring that variable values used as SQL literals and identifiers are properly quoted. The syntax for interpolating a value without any quoting is to prepend the variable name with a colon (`:`). For example,

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

would query the table `my_table`. Note that this may be unsafe: the value of the variable is copied literally, so it can contain unbalanced quotes, or even backslash commands. You must make sure that it makes sense where you put it.

When a value is to be used as an SQL literal or identifier, it is safest to arrange for it to be quoted. To quote the value of a variable as an SQL literal, write a colon followed by the variable name in single quotes. To quote the value as an SQL identifier, write a colon followed by the variable name in double quotes. These constructs deal correctly with quotes and other special characters embedded within the variable value. The previous example would be more safely written this way:

```
testdb=> \set foo 'my_table'
```

```
testdb=> SELECT * FROM :"foo";
```

Variable interpolation will not be performed within quoted SQL literals and identifiers. Therefore, a construction such as `':foo'` doesn't work to produce a quoted literal from a variable's value (and it would be unsafe if it did work, since it wouldn't correctly handle quotes embedded in the value).

One example use of this mechanism is to copy the contents of a file into a table column. First load the file into a variable and then interpolate the variable's value as a quoted string:

```
testdb=> \set content `cat my_file.txt`
testdb=> INSERT INTO my_table VALUES (:'content');
```

(Note that this still won't work if `my_file.txt` contains `NUL` bytes. `psql` does not support embedded `NUL` bytes in variable values.)

Since colons can legally appear in SQL commands, an apparent attempt at interpolation (that is, `:name`, `:'name'`, or `:"name"`) is not replaced unless the named variable is currently set. In any case, you can escape a colon with a backslash to protect it from substitution.

The colon syntax for variables is standard SQL for embedded query languages, such as ECPG. The colon syntaxes for array slices and type casts are Greenplum Database extensions, which can sometimes conflict with the standard usage. The colon-quote syntax for escaping a variable's value as an SQL literal or identifier is a `psql` extension.

Prompting

The prompts `psql` issues can be customized to your preference. The three variables `PROMPT1`, `PROMPT2`, and `PROMPT3` contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when `psql` requests a new command. Prompt 2 is issued when more input is expected during command entry, for example because the command was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you are running an SQL `COPY FROM STDIN` command and you need to type in a row value on the terminal.

The value of the selected prompt variable is printed literally, except where a percent sign (%) is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

%M

The full host name (with domain name) of the database server, or `[local]` if the connection is over a UNIX domain socket, or `[local:/dir/name]`, if the UNIX domain socket is not at the compiled in default location.

%m

The host name of the database server, truncated at the first dot, or `[local]` if the connection is over a UNIX domain socket.

%>

The port number at which the database server is listening.

%n

The database session user name. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

%/

The name of the current database.

%~

Like `%/`, but the output is ~ (tilde) if the database is your default database.

%#

If the session user is a database superuser, then a **#**, otherwise a **>**. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

%R

In prompt 1 normally =, but ^ if in single-line mode, or `!` if the session is disconnected from the database (which can happen if `\connect` fails). In prompt 2 `%R` is replaced by a character that depends on why `psql` expects more input: `-` if the command simply wasn't terminated yet, but `*` if there is an unfinished `/* ... */` comment, a single quote if there is an unfinished quoted string, a double quote if there is an unfinished quoted identifier, a dollar sign if there is an unfinished dollar-quoted string, or `(` if there is an unmatched left parenthesis. In prompt 3 `%R` doesn't produce anything.

%x

Transaction status: an empty string when not in a transaction block, or ***** when in a transaction block, or **!** when in a failed transaction block, or **?** when the transaction state is indeterminate (for example, because there is no connection).

%digits

The character with the indicated octal code is substituted.

%:name:

The value of the `psql` variable name. See "Variables" in *Advanced Features* for details.

%`command`

The output of command, similar to ordinary back-tick substitution.

%[... %]

Prompts may contain terminal control characters which, for example, change the color, background, or style of the prompt text, or change the title of the terminal window. In order for line editing to work properly, these non-printing control characters must be designated as invisible by surrounding them with `%[` and `%]`. Multiple pairs of these may occur within the prompt. For example,

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%#'
```

results in a boldfaced (`1;`) yellow-on-black (`33;40`) prompt on VT100-compatible, color-capable terminals. To insert a percent sign into your prompt, write `%%`. The default prompts are `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

Command-Line Editing

`psql` uses the `readline` library for convenient line editing and retrieval. The command history is automatically saved when `psql` exits and is reloaded when `psql` starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. The queries generated by tab-completion can also interfere with other SQL commands, e.g. `SET TRANSACTION ISOLATION LEVEL`. If for some reason you do not like the tab completion, you can turn it off by putting this in a file named `.inputrc` in your home directory:

```
$if psql
set disable-completion on
$endif
```

Environment

COLUMNS

If `\pset columns` is zero, controls the width for the wrapped format and width for determining if wide output requires the pager or should be switched to the vertical format in expanded auto mode.

PAGER

If the query results do not fit on the screen, they are piped through this command. Typical values are `more` or `less`. The default is platform-dependent. The use of the pager can be disabled by setting *PAGER* to empty, or by using pager-related options of the `\pset` command.

PGDATABASE
PGHOST
PGPORT
PGUSER

Default connection parameters.

PSQL_EDITOR
EDITOR
VISUAL

Editor used by the `\e` and `\ef` commands. The variables are examined in the order listed; the first that is set is used.

The built-in default editors are `vi` on Unix systems and `notepad.exe` on Windows systems.

PSQL_EDITOR_LINENUMBER_ARG

When `\e` or `\ef` is used with a line number argument, this variable specifies the command-line argument used to pass the starting line number to the user's editor. For editors such as Emacs or `vi`, this is a plus sign. Include a trailing space in the value of the variable if there needs to be space between the option name and the line number. Examples:

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line '
```

The default is + on Unix systems (corresponding to the default editor `vi`, and useful for many other common editors); but there is no default on Windows systems.

PSQL_HISTORY

Alternative location for the command history file. Tilde (~) expansion is performed.

PSQLRC

Alternative location of the user's `.psqlrc` file. Tilde (~) expansion is performed.

SHELL

Command executed by the `\!` command.

TMPDIR

Directory for storing temporary files. The default is `/tmp`.

Files

psqlrc and ~/.psqlrc

Unless it is passed an `-X` or `-c` option, `psql` attempts to read and execute commands from the system-wide startup file (`psqlrc`) and then the user's personal startup file (`~/.psqlrc`), after connecting to the database but before accepting normal commands. These files can be used to set up the client and/or the server to taste, typically with `\set` and `SET` commands.

The system-wide startup file is named `psqlrc` and is sought in the installation's "system configuration" directory, which is most reliably identified by running `pg_config --sysconfdir`. By default this directory will be `../etc/` relative to the directory containing the Greenplum Database executables. The name of this directory can be set explicitly via the `PGSYSCONFDIR` environment variable.

The user's personal startup file is named `.psqlrc` and is sought in the invoking user's home directory. On Windows, which lacks such a concept, the personal startup file is named `%APPDATA%\postgresql\psqlrc.conf`. The location of the user's startup file can be set explicitly via the `PSQLRC` environment variable.

Both the system-wide startup file and the user's personal startup file can be made psql-version-specific by appending a dash and the underlying PostgreSQL major or minor release number to the file name, for example `~/.psqlrc-9.4`. The most specific version-matching file will be read in preference to a non-version-specific file.

`.psql_history`

The command-line history is stored in the file `~/.psql_history`, or `%APPDATA%\postgresql\psql_history` on Windows.

The location of the history file can be set explicitly via the `PSQL_HISTORY` environment variable.

Notes

`psql` works best with servers of the same or an older major version. Backslash commands are particularly likely to fail if the server is of a newer version than `psql` itself. However, backslash commands of the `\d` family should work with older server versions, though not necessarily with servers newer than `psql` itself. The general functionality of running SQL commands and displaying query results should also work with servers of a newer major version, but this cannot be guaranteed in all cases.

If you want to use `psql` to connect to several servers of different major versions, it is recommended that you use the newest version of `psql`. Alternatively, you can keep a copy of `psql` from each major version around and be sure to use the version that matches the respective server. But in practice, this additional complication should not be necessary.

Notes for Windows Users

`psql` is built as a console application. Since the Windows console windows use a different encoding than the rest of the system, you must take special care when using 8-bit characters within `psql`. If `psql` detects a problematic console code page, it will warn you at startup. To change the console code page, two things are necessary:

Set the code page by entering:

```
cmd.exe /c chcp 1252
```

`1252` is a character encoding of the Latin alphabet, used by Microsoft Windows for English and some other Western languages. If you are using Cygwin, you can put this command in `/etc/profile`.

Set the console font to Lucida Console, because the raster font does not work with the ANSI code page.

Examples

Start `psql` in interactive mode:

```
psql -p 54321 -U sally mydatabase
```

In `psql` interactive mode, spread a command over several lines of input. Notice the changing prompt:

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text)
testdb-> ;
CREATE TABLE
```

Look at the table definition:

```
testdb=> \d my_table
            Table "my_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 first     | integer | not null default 0
 second    | text    |
```

Run `psql` in non-interactive mode by passing in a file containing SQL commands:

```
psql -f /home/gpadmin/test/myscript.sql
```

reindexdb

Rebuilds indexes in a database.

Synopsis

```
reindexdb [connection-option ...] [--table | -t table ]
        [--index | -i index ] [dbname]

reindexdb [connection-option ...] --all | -a

reindexdb [connection-option ...] --system | -s [dbname]

reindexdb -? | --help

reindexdb -V | --version
```

Description

`reindexdb` is a utility for rebuilding indexes in Greenplum Database.

`reindexdb` is a wrapper around the SQL command `REINDEX`. There is no effective difference between reindexing databases via this utility and via other methods for accessing the server.

Options

-a | --all

Reindex all databases.

[-d] *dbname* | [--dbname=]*dbname*

Specifies the name of the database to be reindexed. If this is not specified and `-all` is not used, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

-e | --echo

Echo the commands that `reindexdb` generates and sends to the server.

-i *index* | --index=*index*

Recreate index only.

-q | --quiet

Do not display a response.

-s | --system

Reindex system catalogs.

-t *table* | --table=*table*

Reindex table only. Multiple tables can be reindexed by writing multiple `-t` switches.

`-V | --version`

Print the `reindexdb` version and exit.

`-? | --help`

Show help about `reindexdb` command line arguments, and exit.

Connection Options

`-h *host* | --host=*host*`

Specifies the host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

`-p *port* | --port=*port*`

Specifies the TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

`-U *username* | --username=*username*`

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system user name.

`-w | --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W | --password`

Force a password prompt.

`--maintenance-db=*dbname*`

Specifies the name of the database to connect to discover what other databases should be reindexed. If not specified, the `postgres` database will be used, and if that does not exist, `template1` will be used.

Notes

`reindexdb` might need to connect several times to the master server, asking for a password each time. It is convenient to have a `~/.pgpass` file in such cases.

Examples

To reindex the database `mydb`:

```
reindexdb mydb
```

To reindex the table `foo` and the index `bar` in a database named `abcd`:

```
reindexdb --table foo --index bar abcd
```

See Also

`REINDEX` in the *Greenplum Database Reference Guide*

vacuumdb

Garbage-collects and analyzes a database.

Synopsis

```
vacuumdb [connection-option...] [--full | -f] [--freeze | -F] [--verbose | -
v]
    [--analyze | -z] [--analyze-only | -Z] [--table | -t table [( column
 [,...] )] ] [dbname]

vacuumdb [connection-option...] [--all | -a] [--full | -f] [-F]
    [--verbose | -v] [--analyze | -z]
    [--analyze-only | -Z]

vacuumdb -? | --help

vacuumdb -V | --version
```

Description

`vacuumdb` is a utility for cleaning a Greenplum Database database. `vacuumdb` will also generate internal statistics used by the Greenplum Database query optimizer.

`vacuumdb` is a wrapper around the SQL command `VACUUM`. There is no effective difference between vacuuming databases via this utility and via other methods for accessing the server.

Options

-a | --all

Vacuums all databases.

[-d] *dbname* | [--dbname=]*dbname*

The name of the database to vacuum. If this is not specified and `-a` (or `--all`) is not used, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

-e | --echo

Echo the commands that `reindexdb` generates and sends to the server.

-f | --full

Selects a full vacuum, which may reclaim more space, but takes much longer and exclusively locks the table.

Warning: A `VACUUM FULL` is not recommended in Greenplum Database.

-F | --freeze

Freeze row transaction information.

-q | --quiet

Do not display a response.

-t *table* [(*column*)] | --table= *table* [(*column*)]

Clean or analyze this table only. Column names may be specified only in conjunction with the `--analyze` or `--analyze-all` options. Multiple tables can be vacuumed by writing multiple `-t` switches. If you specify columns, you probably have to escape the parentheses from the shell.

-v | --verbose

Print detailed information during processing.

-z | --analyze

Collect statistics for use by the query planner.

-Z | --analyze-only

Only calculate statistics for use by the query planner (no vacuum).

-V | --version

Print the vacuumdb version and exit.

-? | --help

Show help about vacuumdb command line arguments, and exit.

Connection Options

-h *host* | --host=*host*

Specifies the host name of the machine on which the Greenplum master database server is running. If not specified, reads from the environment variable PGHOST or defaults to localhost.

-p *port* | --port=*port*

Specifies the TCP port on which the Greenplum master database server is listening for connections. If not specified, reads from the environment variable PGPORT or defaults to 5432.

-U *username* | --username=*username*

The database role name to connect as. If not specified, reads from the environment variable PGUSER or defaults to the current system user name.

-w | --no-password

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a .pgpass file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

-W | --password

Force a password prompt.

--maintenance-db=*dbname*

Specifies the name of the database to connect to discover what other databases should be vacuumed. If not specified, the postgres database will be used, and if that does not exist, template1 will be used.

Notes

vacuumdb might need to connect several times to the master server, asking for a password each time. It is convenient to have a ~/.pgpass file in such cases.

Examples

To clean the database test:

```
vacuumdb test
```

To clean and analyze a database named bigdb:

```
vacuumdb --analyze bigdb
```

To clean a single table foo in a database named mydb, and analyze a single column bar of the table. Note the quotes around the table and column names to escape the parentheses from the shell:

```
vacuumdb --analyze --verbose --table 'foo(bar)' mydb
```

See Also

VACUUM and ANALYZE in the *Greenplum Database Reference Guide*

Additional Supplied Programs

Additional programs available in the Greenplum Database installation.

The following PostgreSQL `contrib` server utility programs are installed:

- *`pg_upgrade`* - Server program to upgrade a Postgres Database server instance.

  Note: `pg_upgrade` is not intended for direct use with Greenplum 6, but will be used by Greenplum upgrade utilities in a future release.
- `pg_upgrade_support` - supporting library for `pg_upgrade`.
- *`pg_xlogdump`* - Server utility program to display a human-readable rendering of the write-ahead log of a Greenplum Database cluster.

Chapter 7

Greenplum Database Reference Guide

Reference information for Greenplum Database systems including SQL commands, system catalogs, environment variables, server configuration parameters, character set support, datatypes, and Greenplum Database extensions.

SQL Commands

The following SQL commands are available in Greenplum Database:

- *CREATE COLLATION*
- *CREATE CONVERSION*
- *CREATE DATABASE*
- *CREATE DOMAIN*
- *CREATE EXTERNAL TABLE*
- *CREATE FOREIGN DATA WRAPPER*
- *CREATE FOREIGN TABLE*
- *CREATE FUNCTION*
- *CREATE GROUP*
- *CREATE INDEX*
- *CREATE LANGUAGE*
- *CREATE OPERATOR*
- *CREATE OPERATOR CLASS*
- *CREATE OPERATOR FAMILY*
- *CREATE PROTOCOL*
- *CREATE RESOURCE GROUP*
- *CREATE RESOURCE QUEUE*
- *CREATE ROLE*
- *CREATE RULE*
- *CREATE SCHEMA*
- *CREATE SEQUENCE*
- *CREATE SERVER*
- *CREATE TABLE*
- *CREATE TABLE AS*
- *CREATE TABLESPACE*
- *CREATE TEXT SEARCH CONFIGURATION*
- *CREATE TEXT SEARCH DICTIONARY*
- *CREATE TEXT SEARCH PARSER*
- *CREATE TEXT SEARCH TEMPLATE*
- *CREATE TYPE*
- *CREATE USER*
- *CREATE USER MAPPING*
- *CREATE VIEW*
- *DEALLOCATE*
- *DECLARE*
- *DELETE*
- *DISCARD*
- *DO*
- *DROP AGGREGATE*
- *DROP CAST*
- *DROP COLLATION*
- *DROP CONVERSION*
- *DROP DATABASE*
- *DROP DOMAIN*
- *DROP EXTERNAL TABLE*
- *DROP FOREIGN DATA WRAPPER*
- *DROP FOREIGN TABLE*
- *DROP FUNCTION*
- *DROP GROUP*
- *DROP INDEX*

- *DROP LANGUAGE*
- *DROP OPERATOR*
- *DROP OPERATOR CLASS*
- *DROP OPERATOR FAMILY*
- *DROP OWNED*
- *DROP PROTOCOL*
- *DROP RESOURCE GROUP*
- *DROP RESOURCE QUEUE*
- *DROP ROLE*
- *DROP RULE*
- *DROP SCHEMA*
- *DROP SEQUENCE*
- *DROP SERVER*
- *DROP TABLE*
- *DROP TABLESPACE*
- *DROP TEXT SEARCH CONFIGURATION*
- *DROP TEXT SEARCH DICTIONARY*
- *DROP TEXT SEARCH PARSER*
- *DROP TEXT SEARCH TEMPLATE*
- *DROP TYPE*
- *DROP USER*
- *DROP USER MAPPING*
- *DROP VIEW*
- *END*
- *EXECUTE*
- *EXPLAIN*
- *FETCH*
- *GRANT*
- *INSERT*
- *LOAD*
- *LOCK*
- *MOVE*
- *PREPARE*
- *REASSIGN OWNED*
- *REINDEX*
- *RELEASE SAVEPOINT*
- *RESET*
- *REVOKE*
- *ROLLBACK*
- *ROLLBACK TO SAVEPOINT*
- *SAVEPOINT*
- *SELECT*
- *SELECT INTO*
- *SET*
- *SET CONSTRAINTS*
- *SET ROLE*
- *SET SESSION AUTHORIZATION*
- *SET TRANSACTION*
- *SHOW*
- *START TRANSACTION*

- *TRUNCATE*
- *UPDATE*
- *VACUUM*
- *VALUES*

** Not implemented in 5.0*

SQL Syntax Summary

ABORT

Aborts the current transaction.

```
ABORT [WORK | TRANSACTION]
```

See *ABORT* for more information.

ALTER AGGREGATE

Changes the definition of an aggregate function

```
ALTER AGGREGATE name ( aggregate_signature )  RENAME TO new_name

ALTER AGGREGATE name ( aggregate_signature ) OWNER TO new_owner

ALTER AGGREGATE name ( aggregate_signature ) SET SCHEMA new_schema
```

See *ALTER AGGREGATE* for more information.

ALTER COLLATION

Changes the definition of a collation.

```
ALTER COLLATION name RENAME TO new_name

ALTER COLLATION name OWNER TO new_owner

ALTER COLLATION name SET SCHEMA new_schema
```

See *ALTER COLLATION* for more information.

ALTER CONVERSION

Changes the definition of a conversion.

```
ALTER CONVERSION name RENAME TO newname

ALTER CONVERSION name OWNER TO newowner

ALTER CONVERSION name SET SCHEMA new_schema
```

See *ALTER CONVERSION* for more information.

ALTER DATABASE

Changes the attributes of a database.

```
ALTER DATABASE name [ WITH CONNECTION LIMIT connlimit ]
```

```
ALTER DATABASE name RENAME TO newname

ALTER DATABASE name OWNER TO new_owner

ALTER DATABASE name SET TABLESPACE new_tablespace

ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name SET parameter FROM CURRENT
ALTER DATABASE name RESET parameter
ALTER DATABASE name RESET ALL
```

See *ALTER DATABASE* for more information.

ALTER DEFAULT PRIVILEGES

Changes default access privileges.

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } target_role [, ...] ]
    [ IN SCHEMA schema_name [, ...] ]
    abbreviated_grant_or_revoke

where abbreviated_grant_or_revoke is one of:

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
 TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
 TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
```

```
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

See *ALTER DEFAULT PRIVILEGES* for more information.

ALTER DOMAIN

Changes the definition of a domain.

```
ALTER DOMAIN name { SET DEFAULT expression | DROP DEFAULT }

ALTER DOMAIN name { SET | DROP } NOT NULL

ALTER DOMAIN name ADD domain_constraint [ NOT VALID ]

ALTER DOMAIN name DROP CONSTRAINT [ IF EXISTS ] constraint_name [RESTRICT |
 CASCADE]

ALTER DOMAIN name RENAME CONSTRAINT constraint_name TO new_constraint_name

ALTER DOMAIN name VALIDATE CONSTRAINT constraint_name

ALTER DOMAIN name OWNER TO new_owner

ALTER DOMAIN name RENAME TO new_name

ALTER DOMAIN name SET SCHEMA new_schema
```

See *ALTER DOMAIN* for more information.

ALTER EXTENSION

Change the definition of an extension that is registered in a Greenplum database.

```
ALTER EXTENSION name UPDATE [ TO new_version ]
ALTER EXTENSION name SET SCHEMA new_schema
ALTER EXTENSION name ADD member_object
ALTER EXTENSION name DROP member_object

where member_object is:

  ACCESS METHOD object_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  CONVERSION object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  [ PROCEDURAL ] LANGUAGE object_name |
  SCHEMA object_name |
```

```
  SEQUENCE object_name |
  SERVER object_name |
  TABLE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TYPE object_name |
  VIEW object_name

and aggregate_signature is:

* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
 ] argtype [ , ... ]
```

See *ALTER EXTENSION* for more information.

ALTER EXTERNAL TABLE

Changes the definition of an external table.

```
ALTER EXTERNAL TABLE name action [, ... ]
```

where *action* is one of:

```
  ADD [COLUMN] new_column type
  DROP [COLUMN] column [RESTRICT|CASCADE]
  ALTER [COLUMN] column TYPE type
  OWNER TO new_owner
```

See *ALTER EXTERNAL TABLE* for more information.

ALTER FOREIGN DATA WRAPPER

Changes the definition of a foreign-data wrapper.

```
ALTER FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]

ALTER FOREIGN DATA WRAPPER name OWNER TO new_owner
ALTER FOREIGN DATA WRAPPER name RENAME TO new_name
```

See *ALTER FOREIGN DATA WRAPPER* for more information.

ALTER FOREIGN TABLE

Changes the definition of a foreign table.

```
ALTER FOREIGN TABLE [ IF EXISTS ] name
    action [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
```

See *ALTER FOREIGN TABLE* for more information.

ALTER FUNCTION

Changes the definition of a function.

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   action [, ... ] [RESTRICT]

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   RENAME TO new_name

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   OWNER TO new_owner

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   SET SCHEMA new_schema
```

See *ALTER FUNCTION* for more information.

ALTER GROUP

Changes a role name or membership.

```
ALTER GROUP groupname ADD USER username [, ... ]

ALTER GROUP groupname DROP USER username [, ... ]

ALTER GROUP groupname RENAME TO newname
```

See *ALTER GROUP* for more information.

ALTER INDEX

Changes the definition of an index.

```
ALTER INDEX [ IF EXISTS ] name RENAME TO new_name

ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name

ALTER INDEX [ IF EXISTS ] name SET ( storage_parameter = value [, ...] )

ALTER INDEX [ IF EXISTS ] name RESET ( storage_parameter  [, ...] )

ALTER INDEX ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
  SET TABLESPACE new_tablespace [ NOWAIT ]
```

See *ALTER INDEX* for more information.

ALTER LANGUAGE

Changes the name of a procedural language.

```
ALTER LANGUAGE name RENAME TO newname
ALTER LANGUAGE name OWNER TO new_owner
```

See *ALTER LANGUAGE* for more information.

ALTER MATERIALIZED VIEW

Changes the definition of a materialized view.

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] name action [, ... ]
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME TO new_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER MATERIALIZED VIEW ALL IN TABLESPACE name [ OWNED BY role_name
 [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]

where action is one of:

    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
 MAIN }
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET ( storage_parameter = value [, ... ] )
    RESET ( storage_parameter [, ... ] )
    OWNER TO new_owner
```

See *ALTER MATERIALIZED VIEW* for more information.

ALTER OPERATOR

Changes the definition of an operator.

```
ALTER OPERATOR name ( {left_type | NONE} , {right_type | NONE} )
   OWNER TO new_owner

ALTER OPERATOR name ( {left_type | NONE} , {right_type | NONE} )
    SET SCHEMA new_schema
```

See *ALTER OPERATOR* for more information.

ALTER OPERATOR CLASS

Changes the definition of an operator class.

```
ALTER OPERATOR CLASS name USING index_method RENAME TO new_name

ALTER OPERATOR CLASS name USING index_method OWNER TO new_owner

ALTER OPERATOR CLASS name USING index_method SET SCHEMA new_schema
```

See *ALTER OPERATOR CLASS* for more information.

ALTER OPERATOR FAMILY

Changes the definition of an operator family.

```
ALTER OPERATOR FAMILY name USING index_method ADD
  {  OPERATOR strategy_number operator_name ( op_type, op_type ) [ FOR
 SEARCH | FOR ORDER BY sort_family_name ]
```

```
    | FUNCTION support_number [ ( op_type [ , op_type ] ) ] funcname
 ( argument_type [, ...] )
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method DROP
  {  OPERATOR strategy_number ( op_type, op_type )
    | FUNCTION support_number [ ( op_type [ , op_type ] )
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method RENAME TO new_name

ALTER OPERATOR FAMILY name USING index_method OWNER TO new_owner

ALTER OPERATOR FAMILY name USING index_method SET SCHEMA new_schema
```

See *ALTER OPERATOR FAMILY* for more information.

ALTER PROTOCOL

Changes the definition of a protocol.

```
ALTER PROTOCOL name RENAME TO newname

ALTER PROTOCOL name OWNER TO newowner
```

See *ALTER PROTOCOL* for more information.

ALTER RESOURCE GROUP

Changes the limits of a resource group.

```
ALTER RESOURCE GROUP name SET group_attribute value
```

See *ALTER RESOURCE GROUP* for more information.

ALTER RESOURCE QUEUE

Changes the limits of a resource queue.

```
ALTER RESOURCE QUEUE name WITH ( queue_attribute=value [, ... ] )
```

See *ALTER RESOURCE QUEUE* for more information.

ALTER ROLE

Changes a database role (user or group).

```
ALTER ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

    SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | CREATEEXTTABLE | NOCREATEEXTTABLE  [ ( attribute='value' [, ...] )
     where attributes and values are:
       type='readable'|'writable'
       protocol='gpfdist'|'http'
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | REPLICATION | NOREPLICATION
```

```
   | CONNECTION LIMIT connlimit
   | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
   | VALID UNTIL 'timestamp'

ALTER ROLE name RENAME TO new_name

ALTER ROLE { name | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE { name | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter FROM CURRENT
ALTER ROLE { name | ALL } [ IN DATABASE database_name ]
 RESET configuration_parameter
ALTER ROLE { name | ALL } [ IN DATABASE database_name ] RESET ALL
ALTER ROLE name RESOURCE QUEUE {queue_name | NONE}
ALTER ROLE name RESOURCE GROUP {group_name | NONE}
```

See *ALTER ROLE* for more information.

ALTER RULE

Changes the definition of a rule.

```
ALTER RULE name ON table_name RENAME TO new_name
```

See *ALTER RULE* for more information.

ALTER SCHEMA

Changes the definition of a schema.

```
ALTER SCHEMA name RENAME TO newname

ALTER SCHEMA name OWNER TO newowner
```

See *ALTER SCHEMA* for more information.

ALTER SEQUENCE

Changes the definition of a sequence generator.

```
ALTER SEQUENCE [ IF EXISTS ] name [INCREMENT [ BY ] increment]
     [MINVALUE minvalue | NO MINVALUE]
     [MAXVALUE maxvalue | NO MAXVALUE]
     [START [ WITH ] start ]
     [RESTART [ [ WITH ] restart] ]
     [CACHE cache] [[ NO ] CYCLE]
     [OWNED BY {table.column | NONE}]

ALTER SEQUENCE [ IF EXISTS ] name OWNER TO new_owner

ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name

ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema
```

See *ALTER SEQUENCE* for more information.

ALTER SERVER

Changes the definition of a foreign server.

```
ALTER SERVER server_name [ VERSION 'new_version' ]
```

```
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]

ALTER SERVER server_name OWNER TO new_owner

ALTER SERVER server_name RENAME TO new_name
```

See *ALTER SERVER* for more information.

ALTER TABLE

Changes the definition of a table.

```
ALTER TABLE [IF EXISTS] [ONLY] name
    action [, ... ]

ALTER TABLE [IF EXISTS] [ONLY] name
    RENAME [COLUMN] column_name TO new_column_name

ALTER TABLE [ IF EXISTS ] [ ONLY ] name
    RENAME CONSTRAINT constraint_name TO new_constraint_name

ALTER TABLE [IF EXISTS] name
    RENAME TO new_name

ALTER TABLE [IF EXISTS] name
    SET SCHEMA new_schema

ALTER TABLE ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]

ALTER TABLE [IF EXISTS] [ONLY] name SET
     WITH (REORGANIZE=true|false)
   | DISTRIBUTED BY ({column_name [opclass]} [, ... ] )
   | DISTRIBUTED RANDOMLY
   | DISTRIBUTED REPLICATED

ALTER TABLE name
   [ ALTER PARTITION { partition_name | FOR (RANK(number))
   | FOR (value) } [...] ] partition_action

where action is one of:

  ADD [COLUMN] column_name data_type [ DEFAULT default_expr ]
      [column_constraint [ ... ]]
      [ COLLATE collation ]
      [ ENCODING ( storage_directive [,...] ) ]
  DROP [COLUMN] [IF EXISTS] column_name [RESTRICT | CASCADE]
  ALTER [COLUMN] column_name [ SET DATA ] TYPE type [COLLATE collation]
 [USING expression]
  ALTER [COLUMN] column_name SET DEFAULT expression
  ALTER [COLUMN] column_name DROP DEFAULT
  ALTER [COLUMN] column_name { SET | DROP } NOT NULL
  ALTER [COLUMN] column_name SET STATISTICS integer
  ALTER [COLUMN] column SET ( attribute_option = value [, ... ] )
  ALTER [COLUMN] column RESET ( attribute_option [, ... ] )
  ADD table_constraint [NOT VALID]
  ADD table_constraint_using_index
  VALIDATE CONSTRAINT constraint_name
  DROP CONSTRAINT [IF EXISTS] constraint_name [RESTRICT | CASCADE]
  DISABLE TRIGGER [trigger_name | ALL | USER]
  ENABLE TRIGGER [trigger_name | ALL | USER]
  CLUSTER ON index_name
  SET WITHOUT CLUSTER
```

```
  SET WITHOUT OIDS
  SET (storage_parameter = value)
  RESET (storage_parameter [, ... ])
  INHERIT parent_table
  NO INHERIT parent_table
  OF type_name
  NOT OF
  OWNER TO new_owner
  SET TABLESPACE new_tablespace
```

See *ALTER TABLE* for more information.

ALTER TABLESPACE

Changes the definition of a tablespace.

```
ALTER TABLESPACE name RENAME TO new_name

ALTER TABLESPACE name OWNER TO new_owner

ALTER TABLESPACE name SET ( tablespace_option = value [, ... ] )

ALTER TABLESPACE name RESET ( tablespace_option [, ... ] )
```

See *ALTER TABLESPACE* for more information.

ALTER TEXT SEARCH CONFIGURATION

Changes the definition of a text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary
 WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ]
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name
ALTER TEXT SEARCH CONFIGURATION name OWNER TO new_owner
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema
```

See *ALTER TEXT SEARCH CONFIGURATION* for more information.

ALTER TEXT SEARCH DICTIONARY

Changes the definition of a text search dictionary.

```
ALTER TEXT SEARCH DICTIONARY name (
    option [ = value ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name
ALTER TEXT SEARCH DICTIONARY name OWNER TO new_owner
ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema
```

See *ALTER TEXT SEARCH DICTIONARY* for more information.

ALTER TEXT SEARCH PARSER

Changes the definition of a text search parser.

```
ALTER TEXT SEARCH PARSER name RENAME TO new_name
ALTER TEXT SEARCH PARSER name SET SCHEMA new_schema
```

See *ALTER TEXT SEARCH PARSER* for more information.

ALTER TEXT SEARCH TEMPLATE

Changes the definition of a text search template.

```
ALTER TEXT SEARCH TEMPLATE name RENAME TO new_name
ALTER TEXT SEARCH TEMPLATE name SET SCHEMA new_schema
```

See *ALTER TEXT SEARCH TEMPLATE* for more information.

ALTER TYPE

Changes the definition of a data type.

```
ALTER TYPE name action [, ... ]
ALTER TYPE name OWNER TO new_owner
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name
 [ CASCADE | RESTRICT ]
ALTER TYPE name RENAME TO new_name
ALTER TYPE name SET SCHEMA new_schema
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE |
 AFTER } existing_enum_value ]
ALTER TYPE name SET DEFAULT ENCODING ( storage_directive )

where action is one of:

  ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE |
 RESTRICT ]
  DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
  ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type
 [ COLLATE collation ] [ CASCADE | RESTRICT ]
```

See *ALTER TYPE* for more information.

ALTER USER

Changes the definition of a database role (user).

```
ALTER USER name RENAME TO newname

ALTER USER name SET config_parameter {TO | =} {value | DEFAULT}

ALTER USER name RESET config_parameter

ALTER USER name RESOURCE QUEUE {queue_name | NONE}

ALTER USER name RESOURCE GROUP {group_name | NONE}

ALTER USER name [ [WITH] option [ ... ] ]
```

See *ALTER USER* for more information.

ALTER USER MAPPING

Changes the definition of a user mapping for a foreign server.

```
ALTER USER MAPPING FOR { username | USER | CURRENT_USER | PUBLIC }
    SERVER servername
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

See *ALTER USER MAPPING* for more information.

ALTER VIEW

Changes properties of a view.

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET
 DEFAULT expression

ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT

ALTER VIEW [ IF EXISTS ] name OWNER TO new_owner

ALTER VIEW [ IF EXISTS ] name RENAME TO new_name

ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema

ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value]
 [, ... ] )

ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

See *ALTER VIEW* for more information.

ANALYZE

Collects statistics about a database.

```
ANALYZE [VERBOSE] [table [ (column [, ...] ) ]]

ANALYZE [VERBOSE] {root_partition|leaf_partition} [ (column [, ...] )]

ANALYZE [VERBOSE] ROOTPARTITION {ALL | root_partition [ (column [, ...] )]}
```

See *ANALYZE* for more information.

BEGIN

Starts a transaction block.

```
BEGIN [WORK | TRANSACTION] [transaction_mode]
```

See *BEGIN* for more information.

CHECKPOINT

Forces a transaction log checkpoint.

```
CHECKPOINT
```

See *CHECKPOINT* for more information.

CLOSE

Closes a cursor.

```
CLOSE cursor_name
```

See *CLOSE* for more information.

CLUSTER

Physically reorders a heap storage table on disk according to an index. Not a recommended operation in Greenplum Database.

```
CLUSTER indexname ON tablename

CLUSTER [VERBOSE] tablename

CLUSTER [VERBOSE]
```

See *CLUSTER* for more information.

COMMENT

Defines or changes the comment of an object.

```
COMMENT ON
{ TABLE object_name |
  COLUMN relation_name.column_name |
  AGGREGATE agg_name (agg_signature) |
  CAST (source_type AS target_type) |
  COLLATION object_name
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  EXTENSION object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION func_name ([[argmode] [argname] argtype [, ...]]) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  [PROCEDURAL] LANGUAGE object_name |
  RESOURCE GROUP object_name |
  RESOURCE QUEUE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  TABLESPACE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name }
IS 'text'
```

See *COMMENT* for more information.

COMMIT

Commits the current transaction.

```
COMMIT [WORK | TRANSACTION]
```

See *COMMIT* for more information.

COPY

Copies data between a file and a table.

```
COPY table_name [(column_name [, ...])]
     FROM {'filename' | PROGRAM 'command' | STDIN}
     [ [ WITH ] ( option [, ...] ) ]
     [ ON SEGMENT ]

COPY { table_name [(column_name [, ...])] | (query)}
     TO {'filename' | PROGRAM 'command' | STDOUT}
     [ [ WITH ] ( option [, ...] ) ]
     [ ON SEGMENT ]
```

See *COPY* for more information.

CREATE AGGREGATE

Defines a new aggregate function.

```
CREATE AGGREGATE name ( [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = statefunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
  )

  CREATE AGGREGATE name ( [ [ argmode ] [ argname ] arg_data_type
 [ , ... ] ]
      ORDER BY [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = statefunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , HYPOTHETICAL ]
  )
```

```
  or the old syntax

  CREATE AGGREGATE name (
    BASETYPE = base_type,
    SFUNC = statefunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
  )
```

See *CREATE AGGREGATE* for more information.

CREATE CAST

Defines a new cast.

```
CREATE CAST (sourcetype AS targettype)
       WITH FUNCTION funcname (argtype [, ...])
       [AS ASSIGNMENT | AS IMPLICIT]

CREATE CAST (sourcetype AS targettype)
       WITHOUT FUNCTION
       [AS ASSIGNMENT | AS IMPLICIT]

CREATE CAST (sourcetype AS targettype)
       WITH INOUT
       [AS ASSIGNMENT | AS IMPLICIT]
```

See *CREATE CAST* for more information.

CREATE COLLATION

Defines a new collation using the specified operating system locale settings, or by copying an existing collation.

```
CREATE COLLATION name (
    [ LOCALE = locale, ]
    [ LC_COLLATE = lc_collate, ]
    [ LC_CTYPE = lc_ctype ])

CREATE COLLATION name FROM existing_collation
```

See *CREATE COLLATION* for more information.

CREATE CONVERSION

Defines a new encoding conversion.

```
CREATE [DEFAULT] CONVERSION name FOR source_encoding TO
     dest_encoding FROM funcname
```

See *CREATE CONVERSION* for more information.

CREATE DATABASE

Creates a new database.

```
CREATE DATABASE name [ [WITH] [OWNER [=] user_name]
                     [TEMPLATE [=] template]
                     [ENCODING [=] encoding]
                     [LC_COLLATE [=] lc_collate]
                     [LC_CTYPE [=] lc_ctype]
                     [TABLESPACE [=] tablespace]
                     [CONNECTION LIMIT [=] connlimit ] ]
```

See *CREATE DATABASE* for more information.

CREATE DOMAIN

Defines a new domain.

```
CREATE DOMAIN name [AS] data_type [DEFAULT expression]
       [ COLLATE collation ]
       [ CONSTRAINT constraint_name
       | NOT NULL | NULL
       | CHECK (expression) [...]]
```

See *CREATE DOMAIN* for more information.

CREATE EXTENSION

Registers an extension in a Greenplum database.

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
  [ WITH ] [ SCHEMA schema_name ]
           [ VERSION version ]
           [ FROM old_version ]
           [ CASCADE ]
```

See *CREATE EXTENSION* for more information.

CREATE EXTERNAL TABLE

Defines a new external table.

```
CREATE [READABLE] EXTERNAL [TEMPORARY | TEMP] TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION ('file://seghost[:port]/path/file' [, ...])
       | ('gpfdist://filehost[:port]/file_pattern[#transform=trans_name]'
            [, ...]
       | ('gpfdists://filehost[:port]/file_pattern[#transform=trans_name]'
            [, ...])
       | ('pxf://path-to-data?PROFILE=profile_name[&SERVER=server_name]
[&custom-option=value[...]]'))
```

```
       | ('s3://S3_endpoint[:port]/bucket_name/[S3_prefix] [region=S3-
region] [config=config_file]')
     [ON MASTER]
     FORMAT 'TEXT'
           [( [HEADER]
              [DELIMITER [AS] 'delimiter' | 'OFF']
              [NULL [AS] 'null string']
              [ESCAPE [AS] 'escape' | 'OFF']
              [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
              [FILL MISSING FIELDS] )]
         | 'CSV'
           [( [HEADER]
              [QUOTE [AS] 'quote']
              [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [FORCE NOT NULL column [, ...]]
              [ESCAPE [AS] 'escape']
              [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
              [FILL MISSING FIELDS] )]
         | 'CUSTOM' (Formatter=<formatter_specifications>)
    [ ENCODING 'encoding' ]
      [ [LOG ERRORS [PERSISTENTLY]] SEGMENT REJECT LIMIT count
      [ROWS | PERCENT] ]

CREATE [READABLE] EXTERNAL WEB [TEMPORARY | TEMP] TABLE table_name
   ( column_name data_type [, ...] | LIKE other_table )
      LOCATION ('http://webhost[:port]/path/file' [, ...])
    | EXECUTE 'command' [ON ALL
                         | MASTER
                         | number_of_segments
                         | HOST ['segment_hostname']
                         | SEGMENT segment_id ]
      FORMAT 'TEXT'
            [( [HEADER]
               [DELIMITER [AS] 'delimiter' | 'OFF']
               [NULL [AS] 'null string']
               [ESCAPE [AS] 'escape' | 'OFF']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
          | 'CSV'
            [( [HEADER]
               [QUOTE [AS] 'quote']
               [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [FORCE NOT NULL column [, ...]]
               [ESCAPE [AS] 'escape']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
          | 'CUSTOM' (Formatter=<formatter specifications>)
     [ ENCODING 'encoding' ]
     [ [LOG ERRORS [PERSISTENTLY]] SEGMENT REJECT LIMIT count
       [ROWS | PERCENT] ]

CREATE WRITABLE EXTERNAL [TEMPORARY | TEMP] TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION('gpfdist://outputhost[:port]/filename[#transform=trans_name]'
          [, ...])
      | ('gpfdists://outputhost[:port]/file_pattern[#transform=trans_name]'
          [, ...])
      FORMAT 'TEXT'
               [( [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [ESCAPE [AS] 'escape' | 'OFF'] )]
          | 'CSV'
```

```
              [([QUOTE [AS] 'quote']
              [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [FORCE QUOTE column [, ...]] | * ]
              [ESCAPE [AS] 'escape'] )]

          | 'CUSTOM' (Formatter=<formatter specifications>)
    [ ENCODING 'write_encoding' ]
    [ DISTRIBUTED BY ({column [opclass]}, [ ... ] ) | DISTRIBUTED RANDOMLY ]

CREATE WRITABLE EXTERNAL [TEMPORARY | TEMP] TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION('s3://S3_endpoint[:port]/bucket_name/[S3_prefix] [region=S3-
region] [config=config_file]')
      [ON MASTER]
      FORMAT 'TEXT'
              [( [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [ESCAPE [AS] 'escape' | 'OFF'] )]
          | 'CSV'
              [([QUOTE [AS] 'quote']
              [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [FORCE QUOTE column [, ...]] | * ]
              [ESCAPE [AS] 'escape'] )]

CREATE WRITABLE EXTERNAL WEB [TEMPORARY | TEMP] TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
    EXECUTE 'command' [ON ALL]
    FORMAT 'TEXT'
              [( [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [ESCAPE [AS] 'escape' | 'OFF'] )]
          | 'CSV'
              [([QUOTE [AS] 'quote']
              [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [FORCE QUOTE column [, ...]] | * ]
              [ESCAPE [AS] 'escape'] )]
           | 'CUSTOM' (Formatter=<formatter specifications>)
    [ ENCODING 'write_encoding' ]
    [ DISTRIBUTED BY ({column [opclass]}, [ ... ] ) | DISTRIBUTED RANDOMLY ]
```

See *CREATE EXTERNAL TABLE* for more information.

CREATE FOREIGN DATA WRAPPER

Defines a new foreign-data wrapper.

```
CREATE FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( [ mpp_execute { 'master' | 'any' | 'all segments' }
 [, ] ] option 'value' [, ... ] ) ]
```

See *CREATE FOREIGN DATA WRAPPER* for more information.

CREATE FOREIGN TABLE

Defines a new foreign table.

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [
```

```
    column_name data_type [ OPTIONS ( option 'value' [, ... ] ) ]
 [ COLLATE collation ] [ column_constraint [ ... ] ]
      [, ... ]
] )
    SERVER server_name
  [ OPTIONS ( [ mpp_execute { 'master' | 'any' | 'all segments' }
 [, ] ] option 'value' [, ... ] ) ]
```

See *CREATE FOREIGN TABLE* for more information.

CREATE FUNCTION

Defines a new function.

```
CREATE [OR REPLACE] FUNCTION name
    ( [ [argmode] [argname] argtype [ { DEFAULT | = } default_expr ]
 [, ...] ] )
      [ RETURNS rettype
        | RETURNS TABLE ( column_name column_type [, ...] ) ]
    { LANGUAGE langname
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [NOT] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    | EXECUTE ON { ANY | MASTER | ALL SEGMENTS | INITPLAN }
    | COST execution_cost
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol' } ...
    [ WITH ({ DESCRIBE = describe_function
           } [, ...] ) ]
```

See *CREATE FUNCTION* for more information.

CREATE GROUP

Defines a new database role.

```
CREATE GROUP name [[WITH] option [ ... ]]
```

See *CREATE GROUP* for more information.

CREATE INDEX

Defines a new index.

```
CREATE [UNIQUE] INDEX [name] ON table_name [USING method]
       ( {column_name | (expression)} [COLLATE parameter] [opclass] [ ASC |
 DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
       [ WITH ( storage_parameter = value [, ... ] ) ]
       [ TABLESPACE tablespace ]
       [ WHERE predicate ]
```

See *CREATE INDEX* for more information.

CREATE LANGUAGE

Defines a new procedural language.

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name
```

```
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ]
   [ VALIDATOR valfunction ]
```

See *CREATE LANGUAGE* for more information.

CREATE MATERIALIZED VIEW

Defines a new materialized view.

```
CREATE MATERIALIZED VIEW table_name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
    [DISTRIBUTED {| BY column [opclass], [ ... ] | RANDOMLY | REPLICATED }]
```

See *CREATE MATERIALIZED VIEW* for more information.

CREATE OPERATOR

Defines a new operator.

```
CREATE OPERATOR name (
       PROCEDURE = funcname
       [, LEFTARG = lefttype] [, RIGHTARG = righttype]
       [, COMMUTATOR = com_op] [, NEGATOR = neg_op]
       [, RESTRICT = res_proc] [, JOIN = join_proc]
       [, HASHES] [, MERGES] )
```

See *CREATE OPERATOR* for more information.

CREATE OPERATOR CLASS

Defines a new operator class.

```
CREATE OPERATOR CLASS name [DEFAULT] FOR TYPE data_type
  USING index_method [ FAMILY family_name ] AS
  { OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ FOR
 SEARCH | FOR ORDER BY sort_family_name ]
  | FUNCTION support_number funcname (argument_type [, ...] )
  | STORAGE storage_type
  } [, ... ]
```

See *CREATE OPERATOR CLASS* for more information.

CREATE OPERATOR FAMILY

Defines a new operator family.

```
CREATE OPERATOR FAMILY name  USING index_method
```

See *CREATE OPERATOR FAMILY* for more information.

CREATE PROTOCOL

Registers a custom data access protocol that can be specified when defining a Greenplum Database external table.

```
CREATE [TRUSTED] PROTOCOL name (
   [readfunc='read_call_handler'] [, writefunc='write_call_handler']
   [, validatorfunc='validate_handler' ])
```

See *CREATE PROTOCOL* for more information.

CREATE RESOURCE GROUP

Defines a new resource group.

```
CREATE RESOURCE GROUP name WITH (group_attribute=value [, ... ])
```

See *CREATE RESOURCE GROUP* for more information.

CREATE RESOURCE QUEUE

Defines a new resource queue.

```
CREATE RESOURCE QUEUE name WITH (queue_attribute=value [, ... ])
```

See *CREATE RESOURCE QUEUE* for more information.

CREATE ROLE

Defines a new database role (user or group).

```
CREATE ROLE name [[WITH] option [ ... ]]
```

See *CREATE ROLE* for more information.

CREATE RULE

Defines a new rewrite rule.

```
CREATE [OR REPLACE] RULE name AS ON event
  TO table_name [WHERE condition]
  DO [ALSO | INSTEAD] { NOTHING | command | (command; command
  ...) }
```

See *CREATE RULE* for more information.

CREATE SCHEMA

Defines a new schema.

```
CREATE SCHEMA schema_name [AUTHORIZATION username]
   [schema_element [ ... ]]

CREATE SCHEMA AUTHORIZATION rolename [schema_element [ ... ]]

CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION user_name ]

CREATE SCHEMA IF NOT EXISTS AUTHORIZATION user_name
```

See *CREATE SCHEMA* for more information.

CREATE SEQUENCE

Defines a new sequence generator.

```
CREATE [TEMPORARY | TEMP] SEQUENCE name
       [INCREMENT [BY] value]
       [MINVALUE minvalue | NO MINVALUE]
       [MAXVALUE maxvalue | NO MAXVALUE]
       [START [ WITH ] start]
       [CACHE cache]
       [[NO] CYCLE]
       [OWNED BY { table.column | NONE }]
```

See *CREATE SEQUENCE* for more information.

CREATE SERVER

Defines a new foreign server.

```
CREATE SERVER server_name [ TYPE 'server_type' ] [ VERSION
 'server_version' ]
    FOREIGN DATA WRAPPER fdw_name
    [ OPTIONS ( [ mpp_execute { 'master' | 'any' | 'all segments' }
 [, ] ] option 'value' [, ... ] ) ]
```

See *CREATE SERVER* for more information.

CREATE TABLE

Defines a new table.

```
CREATE [ [GLOBAL | LOCAL] {TEMPORARY | TEMP } | UNLOGGED] TABLE [IF NOT
 EXISTS]
  table_name (
  [ { column_name data_type [ COLLATE collation ] [column_constraint
 [ ... ] ]
[ ENCODING ( storage_directive [, ...] ) ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    | [ column_reference_storage_directive [, ...]
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [=value] [, ... ] ) ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
[ DISTRIBUTED BY (column [opclass], [ ... ] )
       | DISTRIBUTED RANDOMLY | DISTRIBUTED REPLICATED ]

{ --partitioned table using SUBPARTITION TEMPLATE
[ PARTITION BY partition_type (column)
  {  [ SUBPARTITION BY partition_type (column1)
       SUBPARTITION TEMPLATE ( template_spec ) ]
          [ SUBPARTITION BY partition_type (column2)
            SUBPARTITION TEMPLATE ( template_spec ) ]
              [...]  }
  ( partition_spec ) ]
} |

{ -- partitioned table without SUBPARTITION TEMPLATE
[ PARTITION BY partition_type (column)
```

```
   [ SUBPARTITION BY partition_type (column1) ]
      [ SUBPARTITION BY partition_type (column2) ]
         [...]
  ( partition_spec
     [ ( subpartition_spec_column1
          [ ( subpartition_spec_column2
               [...] ) ] ) ],
  [ partition_spec
     [ ( subpartition_spec_column1
        [ ( subpartition_spec_column2
             [...] ) ] ) ], ]
    [...]
  ) ]
}

CREATE [ [GLOBAL | LOCAL] {TEMPORARY | TEMP} | UNLOGGED ] TABLE [IF NOT
 EXISTS]
   table_name
    OF type_name [ (
  { column_name WITH OPTIONS [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ]
[ WITH ( storage_parameter [=value] [, ... ] ) ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

See *CREATE TABLE* for more information.

CREATE TABLE AS

Defines a new table from the results of a query.

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ]
 TABLE table_name
        [ (column_name [, ...] ) ]
        [ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
        [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
        [ TABLESPACE tablespace_name ]
       AS query
        [ WITH [ NO ] DATA ]
        [ DISTRIBUTED BY (column [, ... ] ) | DISTRIBUTED RANDOMLY |
 DISTRIBUTED REPLICATED ]
```

See *CREATE TABLE AS* for more information.

CREATE TABLESPACE

Defines a new tablespace.

```
CREATE TABLESPACE tablespace_name [OWNER username]  LOCATION '/path/to/dir'
   [WITH (contentID_1='/path/to/dir1'[, contentID_2='/path/to/dir2' ... ])]
```

See *CREATE TABLESPACE* for more information.

CREATE TEXT SEARCH CONFIGURATION

Defines a new text search configuration.

```
CREATE TEXT SEARCH CONFIGURATION name (
    PARSER = parser_name |
```

```
    COPY = source_config
)
```

See *CREATE TEXT SEARCH CONFIGURATION* for more information.

CREATE TEXT SEARCH DICTIONARY

Defines a new text search dictionary.

```
CREATE TEXT SEARCH DICTIONARY name (
    TEMPLATE = template
    [, option = value [, ... ]]
)
```

See *CREATE TEXT SEARCH DICTIONARY* for more information.

CREATE TEXT SEARCH PARSER

Defines a new text search parser.

```
CREATE TEXT SEARCH PARSER name (
    START = start_function ,
    GETTOKEN = gettoken_function ,
    END = end_function ,
    LEXTYPES = lextypes_function
    [, HEADLINE = headline_function ]
)
```

See *CREATE TEXT SEARCH PARSER* for more information.

CREATE TEXT SEARCH TEMPLATE

Defines a new text search template.

```
CREATE TEXT SEARCH TEMPLATE name (
    [ INIT = init_function , ]
    LEXIZE = lexize_function
)
```

See *CREATE TEXT SEARCH TEMPLATE* for more information.

CREATE TYPE

Defines a new data type.

```
CREATE TYPE name AS
    ( attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ]
    [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ]
)

CREATE TYPE name (
    INPUT = input_function,
```

```
    OUTPUT = output_function
    [, RECEIVE = receive_function]
    [, SEND = send_function]
    [, TYPMOD_IN = type_modifier_input_function ]
    [, TYPMOD_OUT = type_modifier_output_function ]
    [, INTERNALLENGTH = {internallength | VARIABLE}]
    [, PASSEDBYVALUE]
    [, ALIGNMENT = alignment]
    [, STORAGE = storage]
    [, LIKE = like_type
    [, CATEGORY = category]
    [, PREFERRED = preferred]
    [, DEFAULT = default]
    [, ELEMENT = element]
    [, DELIMITER = delimiter]
    [, COLLATABLE = collatable]
    [, COMPRESSTYPE = compression_type]
    [, COMPRESSLEVEL = compression_level]
    [, BLOCKSIZE = blocksize] )

CREATE TYPE name
```

See *CREATE TYPE* for more information.

CREATE USER

Defines a new database role with the `LOGIN` privilege by default.

```
CREATE USER name [[WITH] option [ ... ]]
```

See *CREATE USER* for more information.

CREATE USER MAPPING

Defines a new mapping of a user to a foreign server.

```
CREATE USER MAPPING FOR { username | USER | CURRENT_USER | PUBLIC }
    SERVER servername
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

See *CREATE USER MAPPING* for more information.

CREATE VIEW

Defines a new view.

```
CREATE [OR REPLACE] [TEMP | TEMPORARY] [RECURSIVE] VIEW name [ ( column_name
 [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

See *CREATE VIEW* for more information.

DEALLOCATE

Deallocates a prepared statement.

```
DEALLOCATE [PREPARE] name
```

See *DEALLOCATE* for more information.

DECLARE

Defines a cursor.

```
DECLARE name [BINARY] [INSENSITIVE] [NO SCROLL] CURSOR
     [{WITH | WITHOUT} HOLD]
     FOR query [FOR READ ONLY]
```

See *DECLARE* for more information.

DELETE

Deletes rows from a table.

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ONLY] table [[AS] alias]
      [USING usinglist]
      [WHERE condition | WHERE CURRENT OF cursor_name]
      [RETURNING * | output_expression [[AS] output_name] [, …]]
```

See *DELETE* for more information.

DISCARD

Discards the session state.

```
DISCARD { ALL | PLANS | TEMPORARY | TEMP }
```

See *DISCARD* for more information.

DROP AGGREGATE

Removes an aggregate function.

```
DROP AGGREGATE [IF EXISTS] name ( aggregate_signature ) [CASCADE | RESTRICT]
```

See *DROP AGGREGATE* for more information.

DO

Executes an anonymous code block as a transient anonymous function.

```
DO [ LANGUAGE lang_name ] code
```

See *DO* for more information.

DROP CAST

Removes a cast.

```
DROP CAST [IF EXISTS] (sourcetype AS targettype) [CASCADE | RESTRICT]
```

See *DROP CAST* for more information.

DROP COLLATION

Removes a previously defined collation.

```
DROP COLLATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

See *DROP COLLATION* for more information.

DROP CONVERSION

Removes a conversion.

```
DROP CONVERSION [IF EXISTS] name [CASCADE | RESTRICT]
```

See *DROP CONVERSION* for more information.

DROP DATABASE

Removes a database.

```
DROP DATABASE [IF EXISTS] name
```

See *DROP DATABASE* for more information.

DROP DOMAIN

Removes a domain.

```
DROP DOMAIN [IF EXISTS] name [, ...]  [CASCADE | RESTRICT]
```

See *DROP DOMAIN* for more information.

DROP EXTENSION

Removes an extension from a Greenplum database.

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

See *DROP EXTENSION* for more information.

DROP EXTERNAL TABLE

Removes an external table definition.

```
DROP EXTERNAL [WEB] TABLE [IF EXISTS] name [CASCADE | RESTRICT]
```

See *DROP EXTERNAL TABLE* for more information.

DROP FOREIGN DATA WRAPPER

Removes a foreign-data wrapper.

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

See *DROP FOREIGN DATA WRAPPER* for more information.

DROP FOREIGN TABLE

Removes a foreign table.

```
DROP FOREIGN TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

See *DROP FOREIGN TABLE* for more information.

DROP FUNCTION

Removes a function.

```
DROP FUNCTION [IF EXISTS] name ( [ [argmode] [argname] argtype
    [, ...] ] ) [CASCADE | RESTRICT]
```

See *DROP FUNCTION* for more information.

DROP GROUP

Removes a database role.

```
DROP GROUP [IF EXISTS] name [, ...]
```

See *DROP GROUP* for more information.

DROP INDEX

Removes an index.

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] name [, ...] [ CASCADE |
 RESTRICT ]
```

See *DROP INDEX* for more information.

DROP LANGUAGE

Removes a procedural language.

```
DROP [PROCEDURAL] LANGUAGE [IF EXISTS] name [CASCADE | RESTRICT]
```

See *DROP LANGUAGE* for more information.

DROP MATERIALIZED VIEW

Removes a materialized view.

```
DROP MATERIALIZED VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

See *DROP MATERIALIZED VIEW* for more information.

DROP OPERATOR

Removes an operator.

```
DROP OPERATOR [IF EXISTS] name ( {lefttype | NONE} ,
    {righttype | NONE} ) [CASCADE | RESTRICT]
```

See *DROP OPERATOR* for more information.

DROP OPERATOR CLASS

Removes an operator class.

```
DROP OPERATOR CLASS [IF EXISTS] name USING index_method [CASCADE | RESTRICT]
```

See *DROP OPERATOR CLASS* for more information.

DROP OPERATOR FAMILY

Removes an operator family.

```
DROP OPERATOR FAMILY [IF EXISTS] name USING index_method [CASCADE |
 RESTRICT]
```

See *DROP OPERATOR FAMILY* for more information.

DROP OWNED

Removes database objects owned by a database role.

```
DROP OWNED BY name [, ...] [CASCADE | RESTRICT]
```

See *DROP OWNED* for more information.

DROP PROTOCOL

Removes a external table data access protocol from a database.

```
DROP PROTOCOL [IF EXISTS] name
```

See *DROP PROTOCOL* for more information.

DROP RESOURCE GROUP

Removes a resource group.

```
DROP RESOURCE GROUP group_name
```

See *DROP RESOURCE GROUP* for more information.

DROP RESOURCE QUEUE

Removes a resource queue.

```
DROP RESOURCE QUEUE queue_name
```

See *DROP RESOURCE QUEUE* for more information.

DROP ROLE

Removes a database role.

```
DROP ROLE [IF EXISTS] name [, ...]
```

See *DROP ROLE* for more information.

DROP RULE

Removes a rewrite rule.

```
DROP RULE [IF EXISTS] name ON table_name [CASCADE | RESTRICT]
```

See *DROP RULE* for more information.

DROP SCHEMA

Removes a schema.

```
DROP SCHEMA [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

See *DROP SCHEMA* for more information.

DROP SEQUENCE

Removes a sequence.

```
DROP SEQUENCE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

See *DROP SEQUENCE* for more information.

DROP SERVER

Removes a foreign server descriptor.

```
DROP SERVER [ IF EXISTS ] servername [ CASCADE | RESTRICT ]
```

See *DROP SERVER* for more information.

DROP TABLE

Removes a table.

```
DROP TABLE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

See *DROP TABLE* for more information.

DROP TABLESPACE

Removes a tablespace.

```
DROP TABLESPACE [IF EXISTS] tablespacename
```

See *DROP TABLESPACE* for more information.

DROP TEXT SEARCH CONFIGURATION

Removes a text search configuration.

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

See *DROP TEXT SEARCH CONFIGURATION* for more information.

DROP TEXT SEARCH DICTIONARY

Removes a text search dictionary.

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

See *DROP TEXT SEARCH DICTIONARY* for more information.

DROP TEXT SEARCH PARSER

Remove a text search parser.

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

See *DROP TEXT SEARCH PARSER* for more information.

DROP TEXT SEARCH TEMPLATE

Removes a text search template.

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

See *DROP TEXT SEARCH TEMPLATE* for more information.

DROP TYPE

Removes a data type.

```
DROP TYPE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

See *DROP TYPE* for more information.

DROP USER

Removes a database role.

```
DROP USER [IF EXISTS] name [, ...]
```

See *DROP USER* for more information.

DROP USER MAPPING

Removes a user mapping for a foreign server.

```
DROP USER MAPPING [ IF EXISTS ] { username | USER | CURRENT_USER | PUBLIC }
    SERVER servername
```

See *DROP USER MAPPING* for more information.

DROP VIEW

Removes a view.

```
DROP VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

See *DROP VIEW* for more information.

END

Commits the current transaction.

```
END [WORK | TRANSACTION]
```

See *END* for more information.

EXECUTE

Executes a prepared SQL statement.

```
EXECUTE name [ (parameter [, ...] ) ]
```

See *EXECUTE* for more information.

EXPLAIN

Shows the query plan of a statement.

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ANALYZE] [VERBOSE] statement
```

See *EXPLAIN* for more information.

FETCH

Retrieves rows from a query using a cursor.

```
FETCH [ forward_direction { FROM | IN } ] cursor_name
```

See *FETCH* for more information.

GRANT

Defines access privileges.

```
GRANT { {SELECT | INSERT | UPDATE | DELETE | REFERENCES |
TRIGGER | TRUNCATE } [, ...] | ALL [PRIVILEGES] }
    ON { [TABLE] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO { [ GROUP ] role_name | PUBLIC} [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    TO { role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { {USAGE | SELECT | UPDATE} [, ...] | ALL [PRIVILEGES] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { {CREATE | CONNECT | TEMPORARY | TEMP} [, ...] | ALL
[PRIVILEGES] }
    ON DATABASE database_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { EXECUTE | ALL [PRIVILEGES] }
    ON { FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...]
] ) [, ...]
         | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [PRIVILEGES] }
    ON LANGUAGE lang_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [, ...] | ALL [PRIVILEGES] }
    ON SCHEMA schema_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC}  [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [PRIVILEGES] }
    ON TABLESPACE tablespace_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT parent_role [, ...]
    TO member_role [, ...] [WITH ADMIN OPTION]

GRANT { SELECT | INSERT | ALL [PRIVILEGES] }
    ON PROTOCOL protocolname
    TO username
```

See *GRANT* for more information.

INSERT

Creates new rows in a table.

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table [( column [, ...] )]
   {DEFAULT VALUES | VALUES ( {expression | DEFAULT} [, ...] ) [, ...]
 | query}
   [RETURNING * | output_expression [[AS] output_name] [, ...]]
```

See *INSERT* for more information.

LOAD

Loads or reloads a shared library file.

```
LOAD 'filename'
```

See *LOAD* for more information.

LOCK

Locks a table.

```
LOCK [TABLE] [ONLY] name [ * ] [, ...] [IN lockmode MODE] [NOWAIT]
```

See *LOCK* for more information.

MOVE

Positions a cursor.

```
MOVE [ forward_direction [ FROM | IN ] ] cursor_name
```

See *MOVE* for more information.

PREPARE

Prepare a statement for execution.

```
PREPARE name [ (datatype [, ...] ) ] AS statement
```

See *PREPARE* for more information.

REASSIGN OWNED

Changes the ownership of database objects owned by a database role.

```
REASSIGN OWNED BY old_role [, ...] TO new_role
```

See *REASSIGN OWNED* for more information.

REFRESH MATERIALIZED VIEW

Replaces the contents of a materialized view.

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name
    [ WITH [ NO ] DATA ]
```

See *REFRESH MATERIALIZED VIEW* for more information.

REINDEX

Rebuilds indexes.

```
REINDEX {INDEX | TABLE | DATABASE | SYSTEM} name
```

See *REINDEX* for more information.

RELEASE SAVEPOINT

Destroys a previously defined savepoint.

```
RELEASE [SAVEPOINT] savepoint_name
```

See *RELEASE SAVEPOINT* for more information.

RESET

Restores the value of a system configuration parameter to the default value.

```
RESET configuration_parameter

RESET ALL
```

See *RESET* for more information.

REVOKE

Removes access privileges.

```
REVOKE [GRANT OPTION FOR] { {SELECT | INSERT | UPDATE | DELETE
       | REFERENCES | TRIGGER | TRUNCATE } [, ...] | ALL [PRIVILEGES] }

       ON { [TABLE] table_name [, ...]
            | ALL TABLES IN SCHEMA schema_name [, ...] }
       FROM { [ GROUP ] role_name | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [ GRANT OPTION FOR ] { { SELECT | INSERT | UPDATE
       | REFERENCES } ( column_name [, ...] )
       [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
       ON [ TABLE ] table_name [, ...]
       FROM { [ GROUP ]  role_name | PUBLIC } [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [GRANT OPTION FOR] { {USAGE | SELECT | UPDATE} [,...]
       | ALL [PRIVILEGES] }
       ON { SEQUENCE sequence_name [, ...]
            | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {CREATE | CONNECT
       | TEMPORARY | TEMP} [, ...] | ALL [PRIVILEGES] }
       ON DATABASE database_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [ GRANT OPTION FOR ]
       { USAGE | ALL [ PRIVILEGES ] }
       ON DOMAIN domain_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [ CASCADE | RESTRICT ]


REVOKE [ GRANT OPTION FOR ]
       { USAGE | ALL [ PRIVILEGES ] }
       ON FOREIGN DATA WRAPPER fdw_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
       { USAGE | ALL [ PRIVILEGES ] }
       ON FOREIGN SERVER server_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [GRANT OPTION FOR] {EXECUTE | ALL [PRIVILEGES]}
       ON { FUNCTION funcname ( [[argmode] [argname] argtype
                               [, ...]] ) [, ...]
            | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
       FROM { [ GROUP ] role_name | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {USAGE | ALL [PRIVILEGES]}
       ON LANGUAGE langname [, ...]
       FROM { [ GROUP ]  role_name | PUBLIC} [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [GRANT OPTION FOR] { {CREATE | USAGE} [, ...]
```

```
       | ALL [PRIVILEGES] }
       ON SCHEMA schema_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { CREATE | ALL [PRIVILEGES] }
       ON TABLESPACE tablespacename [, ...]
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [ GRANT OPTION FOR ]
       { USAGE | ALL [ PRIVILEGES ] }
       ON TYPE type_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [ADMIN OPTION FOR] parent_role [, ...]
       FROM [ GROUP ] member_role [, ...]
       [CASCADE | RESTRICT]
```

See *REVOKE* for more information.

ROLLBACK

Aborts the current transaction.

```
ROLLBACK [WORK | TRANSACTION]
```

See *ROLLBACK* for more information.

ROLLBACK TO SAVEPOINT

Rolls back the current transaction to a savepoint.

```
ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] savepoint_name
```

See *ROLLBACK TO SAVEPOINT* for more information.

SAVEPOINT

Defines a new savepoint within the current transaction.

```
SAVEPOINT savepoint_name
```

See *SAVEPOINT* for more information.

SELECT

Retrieves rows from a table or view.

```
[ WITH [ RECURSIVE[1] ] with_query [, ...] ]
SELECT [ALL | DISTINCT [ON (expression [, ...])]]
  * | expression [[AS] output_name] [, ...]
  [FROM from_item [, ...]]
  [WHERE condition]
  [GROUP BY grouping_element [, ...]]
  [HAVING condition [, ...]]
  [WINDOW window_name AS (window_definition) [, ...] ]
  [{UNION | INTERSECT | EXCEPT} [ALL | DISTINCT] select]
  [ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}]
 [, ...]]
```

```
  [LIMIT {count | ALL}]
  [OFFSET start [ ROW | ROWS ] ]
  [FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY]
  [FOR {UPDATE | NO KEY UPDATE | SHARE | KEY SHARE} [OF table_name [, ...]]
 [NOWAIT] [...]]

TABLE { [ ONLY ] table_name [ * ] | with_query_name }
```

See *SELECT* for more information.

SELECT INTO

Defines a new table from the results of a query.

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ALL | DISTINCT [ON ( expression [, ...] )]]
    * | expression [AS output_name] [, ...]
    INTO [TEMPORARY | TEMP | UNLOGGED ] [TABLE] new_table
    [FROM from_item [, ...]]
    [WHERE condition]
    [GROUP BY expression [, ...]]
    [HAVING condition [, ...]]
    [{UNION | INTERSECT | EXCEPT} [ALL | DISTINCT ] select]
    [ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST |
 LAST}] [, ...]]
    [LIMIT {count | ALL}]
    [OFFSET start [ ROW | ROWS ] ]
    [FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT]
    [...]]
```

See *SELECT INTO* for more information.

SET

Changes the value of a Greenplum Database configuration parameter.

```
SET [SESSION | LOCAL] configuration_parameter {TO | =} value |
    'value' | DEFAULT}

SET [SESSION | LOCAL] TIME ZONE {timezone | LOCAL | DEFAULT}
```

See *SET* for more information.

SET CONSTRAINTS

Sets constraint check timing for the current transaction.

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

See *SET CONSTRAINTS* for more information.

SET ROLE

Sets the current role identifier of the current session.

```
SET [SESSION | LOCAL] ROLE rolename

SET [SESSION | LOCAL] ROLE NONE
```

```
RESET ROLE
```

See *SET ROLE* for more information.

SET SESSION AUTHORIZATION

Sets the session role identifier and the current role identifier of the current session.

```
SET [SESSION | LOCAL] SESSION AUTHORIZATION rolename

SET [SESSION | LOCAL] SESSION AUTHORIZATION DEFAULT

RESET SESSION AUTHORIZATION
```

See *SET SESSION AUTHORIZATION* for more information.

SET TRANSACTION

Sets the characteristics of the current transaction.

```
SET TRANSACTION [transaction_mode] [READ ONLY | READ WRITE]

SET TRANSACTION SNAPSHOT snapshot_id

SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode
     [READ ONLY | READ WRITE]
     [NOT] DEFERRABLE
```

See *SET TRANSACTION* for more information.

SHOW

Shows the value of a system configuration parameter.

```
SHOW configuration_parameter

SHOW ALL
```

See *SHOW* for more information.

START TRANSACTION

Starts a transaction block.

```
START TRANSACTION [transaction_mode] [READ WRITE | READ ONLY]
```

See *START TRANSACTION* for more information.

TRUNCATE

Empties a table of all rows.

```
TRUNCATE [TABLE] [ONLY] name [ * ] [, ...]
    [ RESTART IDENTITY | CONTINUE IDENTITY ] [CASCADE | RESTRICT]
```

See *TRUNCATE* for more information.

UPDATE

Updates rows of a table.

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ONLY] table [[AS] alias]
   SET {column = {expression | DEFAULT} |
   (column [, ...]) = ({expression | DEFAULT} [, ...])} [, ...]
   [FROM fromlist]
   [WHERE condition | WHERE CURRENT OF cursor_name ]
```

See *UPDATE* for more information.

VACUUM

Garbage-collects and optionally analyzes a database.

```
VACUUM [({ FULL | FREEZE | VERBOSE | ANALYZE } [, ...])] [table [(column
 [, ...] )]]

VACUUM [FULL] [FREEZE] [VERBOSE] [table]

VACUUM [FULL] [FREEZE] [VERBOSE] ANALYZE
              [table [(column [, ...] )]]
```

See *VACUUM* for more information.

VALUES

Computes a set of rows.

```
VALUES ( expression [, ...] ) [, ...]
   [ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
   [LIMIT { count | ALL } ]
   [OFFSET start [ ROW | ROWS ] ]
   [FETCH { FIRST | NEXT } [count ] { ROW | ROWS } ONLY ]
```

See *VALUES* for more information.

ABORT

Aborts the current transaction.

Synopsis

```
ABORT [WORK | TRANSACTION]
```

Description

ABORT rolls back the current transaction and causes all the updates made by the transaction to be discarded. This command is identical in behavior to the standard SQL command *ROLLBACK*, and is present only for historical reasons.

Parameters

WORK

TRANSACTION

Optional key words. They have no effect.

Notes

Use `COMMIT` to successfully terminate a transaction.

Issuing `ABORT` when not inside a transaction does no harm, but it will provoke a warning message.

Compatibility

This command is a Greenplum Database extension present for historical reasons. `ROLLBACK` is the equivalent standard SQL command.

See Also

`BEGIN`, *`COMMIT`*, *`ROLLBACK`*

ALTER AGGREGATE

Changes the definition of an aggregate function

Synopsis

```
ALTER AGGREGATE name ( aggregate_signature )  RENAME TO new_name

ALTER AGGREGATE name ( aggregate_signature ) OWNER TO new_owner

ALTER AGGREGATE name ( aggregate_signature ) SET SCHEMA new_schema
```

where *aggregate_signature* is:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
 ] argtype [ , ... ]
```

Description

`ALTER AGGREGATE` changes the definition of an aggregate function.

You must own the aggregate function to use `ALTER AGGREGATE`. To change the schema of an aggregate function, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the aggregate function's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the aggregate function. However, a superuser can alter ownership of any aggregate function anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing aggregate function.

argmode

The mode of an argument: `IN` or `VARIADIC`. If omitted, the default is `IN`.

argname

The name of an argument. Note that `ALTER AGGREGATE` does not actually pay any attention to argument names, since only the argument data types are needed to determine the aggregate function's identity.

argtype

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write * in place of the list of input data types. To reference an ordered-set aggregate function, write ORDER BY between the direct and aggregated argument specifications.

new_name

The new name of the aggregate function.

new_owner

The new owner of the aggregate function.

new_schema

The new schema for the aggregate function.

Notes

The recommended syntax for referencing an ordered-set aggregate is to write ORDER BY between the direct and aggregated argument specifications, in the same style as in *CREATE AGGREGATE*. However, it will also work to omit ORDER BY and just run the direct and aggregated argument specifications into a single list. In this abbreviated form, if VARIADIC "any" was used in both the direct and aggregated argument lists, write VARIADIC "any" only once.

Examples

To rename the aggregate function myavg for type integer to my_average:

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

To change the owner of the aggregate function myavg for type integer to joe:

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

To move the aggregate function myavg for type integer into schema myschema:

```
ALTER AGGREGATE myavg(integer) SET SCHEMA myschema;
```

Compatibility

There is no ALTER AGGREGATE statement in the SQL standard.

See Also

CREATE AGGREGATE, *DROP AGGREGATE*

ALTER COLLATION

Changes the definition of a collation.

Synopsis

```
ALTER COLLATION name RENAME TO new_name

ALTER COLLATION name OWNER TO new_owner

ALTER COLLATION name SET SCHEMA new_schema
```

Parameters

name

The name (optionally schema-qualified) of an existing collation.

new_name

The new name of the collation.

new_owner

The new owner of the collation.

new_schema

The new schema for the collation.

Description

You must own the collation to use `ALTER COLLATION`. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the collation's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the collation. However, a superuser can alter ownership of any collation anyway.)

Examples

To rename the collation *de_DE* to `german`:

```
ALTER COLLATION "de_DE" RENAME TO german;
```

To change the owner of the collation `en_US` to `joe`:

```
ALTER COLLATION "en_US" OWNER TO joe;
```

Compatibility

There is no `ALTER COLLATION` statement in the SQL standard.

See Also

CREATE COLLATION, *DROP COLLATION*

ALTER CONVERSION

Changes the definition of a conversion.

Synopsis

```
ALTER CONVERSION name RENAME TO newname

ALTER CONVERSION name OWNER TO newowner

ALTER CONVERSION name SET SCHEMA new_schema
```

Description

`ALTER CONVERSION` changes the definition of a conversion.

You must own the conversion to use `ALTER CONVERSION`. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the conversion's schema. (These restrictions enforce that altering the owner does not do anything you could not do by

dropping and recreating the conversion. However, a superuser can alter ownership of any conversion anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing conversion.

newname

The new name of the conversion.

newowner

The new owner of the conversion.

new_schema

The new schema for the conversion.

Examples

To rename the conversion `iso_8859_1_to_utf8` to `latin1_to_unicode`:

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO
latin1_to_unicode;
```

To change the owner of the conversion `iso_8859_1_to_utf8` to `joe`:

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

Compatibility

There is no `ALTER CONVERSION` statement in the SQL standard.

See Also

`CREATE CONVERSION`, *`DROP CONVERSION`*

ALTER DATABASE

Changes the attributes of a database.

Synopsis

```
ALTER DATABASE name [ WITH CONNECTION LIMIT connlimit ]

ALTER DATABASE name RENAME TO newname

ALTER DATABASE name OWNER TO new_owner

ALTER DATABASE name SET TABLESPACE new_tablespace

ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name SET parameter FROM CURRENT
ALTER DATABASE name RESET parameter
ALTER DATABASE name RESET ALL
```

Description

`ALTER DATABASE` changes the attributes of a database.

The first form changes the allowed connection limit for a database. Only the database owner or a superuser can change this setting.

The second form changes the name of the database. Only the database owner or a superuser can rename a database; non-superuser owners must also have the `CREATEDB` privilege. You cannot rename the current database. Connect to a different database first.

The third form changes the owner of the database. To alter the owner, you must own the database and also be a direct or indirect member of the new owning role, and you must have the `CREATEDB` privilege. (Note that superusers have all these privileges automatically.)

The fourth form changes the default tablespace of the database. Only the database owner or a superuser can do this; you must also have create privilege for the new tablespace. This command physically moves any tables or indexes in the database's old default tablespace to the new tablespace. Note that tables and indexes in non-default tablespaces are not affected.

The remaining forms change the session default for a configuration parameter for a Greenplum database. Whenever a new session is subsequently started in that database, the specified value becomes the session default value. The database-specific default overrides whatever setting is present in the server configuration file (`postgresql.conf`). Only the database owner or a superuser can change the session defaults for a database. Certain parameters cannot be set this way, or can only be set by a superuser.

Parameters

name

The name of the database whose attributes are to be altered.

connlimit

The maximum number of concurrent connections possible. The default of -1 means there is no limitation.

parameter value

Set this database's session default for the specified configuration parameter to the given value. If value is `DEFAULT` or, equivalently, `RESET` is used, the database-specific setting is removed, so the system-wide default setting will be inherited in new sessions. Use `RESET ALL` to clear all database-specific settings. See *Server Configuration Parameters* for information about all user-settable configuration parameters.

newname

The new name of the database.

new_owner

The new owner of the database.

new_tablespace

The new default tablespace of the database.

Notes

It is also possible to set a configuration parameter session default for a specific role (user) rather than to a database. Role-specific settings override database-specific ones if there is a conflict. See `ALTER ROLE`.

Examples

To set the default schema search path for the `mydatabase` database:

```
ALTER DATABASE mydatabase SET search_path TO myschema,
public, pg_catalog;
```

Compatibility

The ALTER DATABASE statement is a Greenplum Database extension.

See Also

CREATE DATABASE, *DROP DATABASE*, *SET*, *CREATE TABLESPACE*

ALTER DEFAULT PRIVILEGES

Changes default access privileges.

Synopsis

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } target_role [, ...] ]
    [ IN SCHEMA schema_name [, ...] ]
    abbreviated_grant_or_revoke

where abbreviated_grant_or_revoke is one of:

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
 TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
 TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

Description

`ALTER DEFAULT PRIVILEGES` allows you to set the privileges that will be applied to objects created in the future. (It does not affect privileges assigned to already-existing objects.) Currently, only the privileges for tables (including views and foreign tables), sequences, functions, and types (including domains) can be altered.

You can change default privileges only for objects that will be created by yourself or by roles that you are a member of. The privileges can be set globally (i.e., for all objects created in the current database), or just for objects created in specified schemas. Default privileges that are specified per-schema are added to whatever the global default privileges are for the particular object type.

As explained under *GRANT*, the default privileges for any object type normally grant all grantable permissions to the object owner, and may grant some privileges to `PUBLIC` as well. However, this behavior can be changed by altering the global default privileges with `ALTER DEFAULT PRIVILEGES`.

Parameters

target_role

The name of an existing role of which the current role is a member. If `FOR ROLE` is omitted, the current role is assumed.

schema_name

The name of an existing schema. If specified, the default privileges are altered for objects later created in that schema. If `IN SCHEMA` is omitted, the global default privileges are altered.

role_name

The name of an existing role to grant or revoke privileges for. This parameter, and all the other parameters in *abbreviated_grant_or_revoke*, act as described under *GRANT* or *REVOKE*, except that one is setting permissions for a whole class of objects rather than specific named objects.

Notes

Use *psql*'s `\ddp` command to obtain information about existing assignments of default privileges. The meaning of the privilege values is the same as explained for `\dp` under *GRANT*.

If you wish to drop a role for which the default privileges have been altered, it is necessary to reverse the changes in its default privileges or use `DROP OWNED BY` to get rid of the default privileges entry for the role.

Examples

Grant SELECT privilege to everyone for all tables (and views) you subsequently create in schema `myschema`, and allow role `webuser` to INSERT into them too:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO
 PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO
 webuser;
```

Undo the above, so that subsequently-created tables won't have any more permissions than normal:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM
 PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM
 webuser;
```

Remove the public EXECUTE permission that is normally granted on functions, for all functions subsequently created by role `admin`:

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM
 PUBLIC;
```

Compatibility

There is no `ALTER DEFAULT PRIVILEGES` statement in the SQL standard.

See Also

GRANT, *REVOKE*

ALTER DOMAIN

Changes the definition of a domain.

Synopsis

```
ALTER DOMAIN name { SET DEFAULT expression | DROP DEFAULT }

ALTER DOMAIN name { SET | DROP } NOT NULL

ALTER DOMAIN name ADD domain_constraint [ NOT VALID ]

ALTER DOMAIN name DROP CONSTRAINT [ IF EXISTS ] constraint_name [RESTRICT |
 CASCADE]

ALTER DOMAIN name RENAME CONSTRAINT constraint_name TO new_constraint_name

ALTER DOMAIN name VALIDATE CONSTRAINT constraint_name

ALTER DOMAIN name OWNER TO new_owner

ALTER DOMAIN name RENAME TO new_name

ALTER DOMAIN name SET SCHEMA new_schema
```

Description

`ALTER DOMAIN` changes the definition of an existing domain. There are several sub-forms:

- **SET/DROP DEFAULT** — These forms set or remove the default value for a domain. Note that defaults only apply to subsequent `INSERT` commands. They do not affect rows already in a table using the domain.
- **SET/DROP NOT NULL** — These forms change whether a domain is marked to allow `NULL` values or to reject `NULL` values. You may only `SET NOT NULL` when the columns using the domain contain no null values.

- **ADD *domain_constraint* [NOT VALID]** — This form adds a new constraint to a domain using the same syntax as `CREATE DOMAIN`. When a new constraint is added to a domain, all columns using that domain will be checked against the newly added constraint. These checks can be suppressed by adding the new constraint using the `NOT VALID` option; the constraint can later be made valid using `ALTER DOMAIN ... VALIDATE CONSTRAINT`. Newly inserted or updated rows are always checked against all constraints, even those marked `NOT VALID`. `NOT VALID` is only accepted for `CHECK` constraints.
- **DROP CONSTRAINT [IF EXISTS]** — This form drops constraints on a domain. If `IF EXISTS` is specified and the constraint does not exist, no error is thrown. In this case a notice is issued instead.
- **RENAME CONSTRAINT** — This form changes the name of a constraint on a domain.
- **VALIDATE CONSTRAINT** — This form validates a constraint previously added as `NOT VALID`, that is, verify that all data in columns using the domain satisfy the specified constraint.
- **OWNER** — This form changes the owner of the domain to the specified user.
- **RENAME** — This form changes the name of the domain.
- **SET SCHEMA** — This form changes the schema of the domain. Any constraints associated with the domain are moved into the new schema as well.

You must own the domain to use `ALTER DOMAIN`. To change the schema of a domain, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the domain's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the domain. However, a superuser can alter ownership of any domain anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing domain to alter.

domain_constraint

New domain constraint for the domain.

constraint_name

Name of an existing constraint to drop or rename.

NOT VALID

Do not verify existing column data for constraint validity.

CASCADE

Automatically drop objects that depend on the constraint.

RESTRICT

Refuse to drop the constraint if there are any dependent objects. This is the default behavior.

new_name

The new name for the domain.

new_constraint_name

The new name for the constraint.

new_owner

The user name of the new owner of the domain.

new_schema

The new schema for the domain.

Examples

To add a `NOT NULL` constraint to a domain:

```
ALTER DOMAIN zipcode SET NOT NULL;
```

To remove a `NOT NULL` constraint from a domain:

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

To add a check constraint to a domain:

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

To remove a check constraint from a domain:

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

To rename a check constraint on a domain:

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO zip_check;
```

To move the domain into a different schema:

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

Compatibility

`ALTER DOMAIN` conforms to the SQL standard, except for the `OWNER`, `RENAME`, `SET SCHEMA`, and `VALIDATE CONSTRAINT` variants, which are Greenplum Database extensions. The `NOT VALID` clause of the `ADD CONSTRAINT` variant is also a Greenplum Database extension.

See Also

`CREATE DOMAIN`, *`DROP DOMAIN`*

ALTER EXTENSION

Change the definition of an extension that is registered in a Greenplum database.

Synopsis

```
ALTER EXTENSION name UPDATE [ TO new_version ]
ALTER EXTENSION name SET SCHEMA new_schema
ALTER EXTENSION name ADD member_object
ALTER EXTENSION name DROP member_object

where member_object is:

  ACCESS METHOD object_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  CONVERSION object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
```

```
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  [ PROCEDURAL ] LANGUAGE object_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  TABLE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TYPE object_name |
  VIEW object_name

and aggregate_signature is:

* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
 ] argtype [ , ... ]
```

Description

`ALTER EXTENSION` changes the definition of an installed extension. These are the subforms:

`UPDATE`

This form updates the extension to a newer version. The extension must supply a suitable update script (or series of scripts) that can modify the currently-installed version into the requested version.

`SET SCHEMA`

This form moves the extension member objects into another schema. The extension must be *relocatable*.

`ADD` *`member_object`*

This form adds an existing object to the extension. This is useful in extension update scripts. The added object is treated as a member of the extension. The object can only be dropped by dropping the extension.

`DROP` *`member_object`*

This form removes a member object from the extension. This is mainly useful in extension update scripts. The object is not dropped, only disassociated from the extension.

See *Packaging Related Objects into an Extension* for more information about these operations.

You must own the extension to use `ALTER EXTENSION`. The `ADD` and `DROP` forms also require ownership of the object that is being added or dropped.

Parameters

`name`

The name of an installed extension.

`new_version`

The new version of the extension. The *new_version* can be either an identifier or a string literal. If not specified, the command attempts to update to the default version in the extension control file.

new_schema

The new schema for the extension.

object_name
aggregate_name
function_name
operator_name

The name of an object to be added to or removed from the extension. Names of tables, aggregates, domains, foreign tables, functions, operators, operator classes, operator families, sequences, text search objects, types, and views can be schema-qualified.

source_type

The name of the source data type of the cast.

target_type

The name of the target data type of the cast.

argmode

The mode of a function or aggregate argument: `IN`, `OUT`, `INOUT`, or `VARIADIC`. The default is `IN`.

The command ignores the `OUT` arguments. Only the input arguments are required to determine the function identity. It is sufficient to list the `IN`, `INOUT`, and `VARIADIC` arguments.

argname

The name of a function or aggregate argument.

The command ignores argument names, since only the argument data types are required to determine the function identity.

argtype

The data type of a function or aggregate argument.

left_type
right_type

The data types of the operator's arguments (optionally schema-qualified) . Specify `NONE` for the missing argument of a prefix or postfix operator.

PROCEDURAL

This is a noise word.

type_name

The name of the data type of the transform.

lang_name

The name of the language of the transform.

Examples

To update the hstore extension to version 2.0:

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

To change the schema of the `hstore` extension to `utils`:

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

To add an existing function to the `hstore` extension:

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);
```

Compatibility

`ALTER EXTENSION` is a Greenplum Database extension.

See Also

`CREATE EXTENSION`, *`DROP EXTENSION`*

ALTER EXTERNAL TABLE

Changes the definition of an external table.

Synopsis

```
ALTER EXTERNAL TABLE name action [, ... ]
```

where *action* is one of:

```
  ADD [COLUMN] new_column type
  DROP [COLUMN] column [RESTRICT|CASCADE]
  ALTER [COLUMN] column TYPE type
  OWNER TO new_owner
```

Description

`ALTER EXTERNAL TABLE` changes the definition of an existing external table. These are the supported `ALTER EXTERNAL TABLE` actions:

- **ADD COLUMN** — Adds a new column to the external table definition.
- **DROP COLUMN** — Drops a column from the external table definition. If you drop readable external table columns, it only changes the table definition in Greenplum Database. The `CASCADE` keyword is required if anything outside the table depends on the column, such as a view that references the column.
- **ALTER COLUMN TYPE** — Changes the data type of a table column.
- **OWNER** — Changes the owner of the external table to the specified user.

Use the *`ALTER TABLE`* command to perform these actions on an external table.

- Set (change) the table schema.
- Rename the table.
- Rename a table column.

You must own the external table to use `ALTER EXTERNAL TABLE` or `ALTER TABLE`. To change the schema of an external table, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the external table's schema. A superuser has these privileges automatically.

Changes to the external table definition with either `ALTER EXTERNAL TABLE` or `ALTER TABLE` do not affect the external data.

The `ALTER EXTERNAL TABLE` and `ALTER TABLE` commands cannot modify the type external table (read, write, web), the table `FORMAT` information, or the location of the external data. To modify this information, you must drop and recreate the external table definition.

Parameters

name

The name (possibly schema-qualified) of an existing external table definition to alter.

column

Name of an existing column.

new_column

Name of a new column.

type

Data type of the new column, or new data type for an existing column.

new_owner

The role name of the new owner of the external table.

CASCADE

Automatically drop objects that depend on the dropped column, such as a view that references the column.

RESTRICT

Refuse to drop the column or constraint if there are any dependent objects. This is the default behavior.

Examples

Add a new column to an external table definition:

```
ALTER EXTERNAL TABLE ext_expenses ADD COLUMN manager text;
```

Change the owner of an external table:

```
ALTER EXTERNAL TABLE ext_data OWNER TO jojo;
```

Change the data type of an external table:

```
ALTER EXTERNAL TABLE ext_leads ALTER COLUMN acct_code TYPE integer
```

Compatibility

`ALTER EXTERNAL TABLE` is a Greenplum Database extension. There is no `ALTER EXTERNAL TABLE` statement in the SQL standard or regular PostgreSQL.

See Also

CREATE EXTERNAL TABLE, *DROP EXTERNAL TABLE*, *ALTER TABLE*

ALTER FOREIGN DATA WRAPPER

Changes the definition of a foreign-data wrapper.

Synopsis

```
ALTER FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]

ALTER FOREIGN DATA WRAPPER name OWNER TO new_owner
```

```
ALTER FOREIGN DATA WRAPPER name RENAME TO new_name
```

Description

`ALTER FOREIGN DATA WRAPPER` changes the definition of a foreign-data wrapper. The first form of the command changes the support functions or generic options of the foreign-data wrapper. Greenplum Database requires at least one clause. The second and third forms of the command change the owner or name of the foreign-data wrapper.

Only superusers can alter foreign-data wrappers. Additionally, only superusers can own foreign-data wrappers

Parameters

name

The name of an existing foreign-data wrapper.

HANDLER *handler_function*

Specifies a new handler function for the foreign-data wrapper.

NO HANDLER

Specifies that the foreign-data wrapper should no longer have a handler function.

> **Note:** You cannot access a foreign table that uses a foreign-data wrapper with no handler.

VALIDATOR *validator_function*

Specifies a new validator function for the foreign-data wrapper.

Options to the foreign-data wrapper, servers, and user mappings may become invalid when you change the validator function. You must make sure that these options are correct before using the modified foreign-data wrapper. Note that Greenplum Database checks any options specified in this `ALTER FOREIGN DATA WRAPPER` command using the new validator.

NO VALIDATOR

Specifies that the foreign-data wrapper should no longer have a validator function.

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

Change the foreign-data wrapper's options. `ADD`, `SET`, and `DROP` specify the action to perform. If no operation is explicitly specified, the default operation is `ADD`. Option names must be unique. Greenplum Database validates names and values using the foreign-data wrapper's validator function, if any.

OWNER TO *new_owner*

Specifies the new owner of the foreign-data wrapper. Only superusers can own foreign-data wrappers.

RENAME TO *new_name*

Specifies the new name of the foreign-data wrapper.

Examples

Change the definition of a foreign-data wrapper named `dbi` by adding a new option named `foo`, and removing the option named `bar`:

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP 'bar');
```

Change the validator function for a foreign-data wrapper named `dbi` to `bob.myvalidator`:

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

Compatibility

`ALTER FOREIGN DATA WRAPPER` conforms to ISO/IEC 9075-9 (SQL/MED), with the exception that the `HANDLER`, `VALIDATOR`, `OWNER TO`, and `RENAME TO` clauses are Greenplum Database extensions.

See Also

CREATE FOREIGN DATA WRAPPER, *DROP FOREIGN DATA WRAPPER*

ALTER FOREIGN TABLE

Changes the definition of a foreign table.

Synopsis

```
ALTER FOREIGN TABLE [ IF EXISTS ] name
    action [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
```

where *action* is one of:

```
    ADD [ COLUMN ] column_name column_type [ COLLATE collation ]
 [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name OPTIONS ( [ ADD | SET | DROP ] option
 ['value'] [, ... ])
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    OWNER TO new_owner
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

Description

`ALTER FOREIGN TABLE` changes the definition of an existing foreign table. There are several subforms of the command:

`ADD COLUMN`

This form adds a new column to the foreign table, using the same syntax as *CREATE FOREIGN TABLE*. Unlike the case when you add a column to a regular table, nothing

happens to the underlying storage: this action simply declares that some new column is now accessible through the foreign table.

DROP COLUMN [IF EXISTS]

This form drops a column from a foreign table. You must specify CASCADE if any objects outside of the table depend on the column; for example, views. If you specify IF EXISTS and the column does not exist, no error is thrown. Greenplum Database issues a notice instead.

IF EXISTS

If you specify IF EXISTS and the foreign table does not exist, no error is thrown. Greenplum Database issues a notice instead.

SET DATA TYPE

This form changes the type of a column of a foreign table.

SET/DROP DEFAULT

These forms set or remove the default value for a column. Default values apply only in subsequent INSERT or UPDATE commands; they do not cause rows already in the table to change.

SET/DROP NOT NULL

Mark a column as allowing, or not allowing, null values.

SET STATISTICS

This form sets the per-column statistics-gathering target for subsequent ANALYZE operations. See the similar form of *ALTER TABLE* for more details.

SET (*attribute_option* = *value* [, ...]])
RESET (*attribute_option* [, ...])

This form sets or resets per-attribute options. See the similar form of *ALTER TABLE* for more details.

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

These forms configure the firing of trigger(s) belonging to the foreign table. See the similar form of *ALTER TABLE* for more details.

OWNER

This form changes the owner of the foreign table to the specified user.

RENAME

The RENAME forms change the name of a foreign table or the name of an individual column in a foreign table.

SET SCHEMA

This form moves the foreign table into another schema.

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

Change options for the foreign table. ADD, SET, and DROP specify the action to perform. If no operation is explicitly specified, the default operation is ADD. Option names must be unique. Greenplum Database validates names and values using the server's foreign-data wrapper.

You can combine all of the actions except RENAME and SET SCHEMA into a list of multiple alterations for Greenplum Database to apply in parallel. For example, it is possible to add several columns and/or alter the type of several columns in a single command.

You must own the table to use `ALTER FOREIGN TABLE`. To change the schema of a foreign table, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the table's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the table. However, a superuser can alter ownership of any table anyway.) To add a column or to alter a column type, you must also have `USAGE` privilege on the data type.

Parameters

name

The name (possibly schema-qualified) of an existing foreign table to alter.

column_name

The name of a new or existing column.

new_column_name

The new name for an existing column.

new_name

The new name for the foreign table.

data_type

The data type of the new column, or new data type for an existing column.

CASCADE

Automatically drop objects that depend on the dropped column (for example, views referencing the column).

RESTRICT

Refuse to drop the column if there are any dependent objects. This is the default behavior.

trigger_name

Name of a single trigger to disable or enable.

ALL

Disable or enable all triggers belonging to the foreign table. (This requires superuser privilege if any of the triggers are internally generated triggers. The core system does not add such triggers to foreign tables, but add-on code could do so.)

USER

Disable or enable all triggers belonging to the foreign table except for internally generated triggers.

new_owner

The user name of the new owner of the foreign table.

new_schema

The name of the schema to which the foreign table will be moved.

Notes

The key word `COLUMN` is noise and can be omitted.

Consistency with the foreign server is not checked when a column is added or removed with `ADD COLUMN` or `DROP COLUMN`, a `NOT NULL` constraint is added, or a column type is changed with `SET DATA TYPE`. It is your responsibility to ensure that the table definition matches the remote side.

Refer to *`CREATE FOREIGN TABLE`* for a further description of valid parameters.

Examples

To mark a column as not-null:

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

To change the options of a foreign table:

```
ALTER FOREIGN TABLE myschema.distributors
    OPTIONS (ADD opt1 'value', SET opt2 'value2', DROP opt3 'value3');
```

Compatibility

The forms `ADD`, `DROP`, and `SET DATA TYPE` conform with the SQL standard. The other forms are Greenplum Database extensions of the SQL standard. The ability to specify more than one manipulation in a single `ALTER FOREIGN TABLE` command is also a Greenplum Database extension.

You can use `ALTER FOREIGN TABLE ... DROP COLUMN` to drop the only column of a foreign table, leaving a zero-column table. This is an extension of SQL, which disallows zero-column foreign tables.

See Also

ALTER TABLE, *CREATE FOREIGN TABLE*, *DROP FOREIGN TABLE*

ALTER FUNCTION

Changes the definition of a function.

Synopsis

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   action [, ... ] [RESTRICT]

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   RENAME TO new_name

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   OWNER TO new_owner

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   SET SCHEMA new_schema
```

where *action* is one of:

```
{CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
{IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF}
{[EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER}
EXECUTE ON { ANY | MASTER | ALL SEGMENTS | INITPLAN }
COST execution_cost
SET configuration_parameter { TO | = } { value | DEFAULT }
SET configuration_parameter FROM CURRENT
RESET configuration_parameter
RESET ALL
```

Description

`ALTER FUNCTION` changes the definition of a function.

You must own the function to use `ALTER FUNCTION`. To change a function's schema, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect

member of the new owning role, and that role must have CREATE privilege on the function's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the function. However, a superuser can alter ownership of any function anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: either IN, OUT, INOUT, or VARIADIC. If omitted, the default is IN. Note that ALTER FUNCTION does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the IN, INOUT, and VARIADIC arguments.

argname

The name of an argument. Note that ALTER FUNCTION does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

new_name

The new name of the function.

new_owner

The new owner of the function. Note that if the function is marked SECURITY DEFINER, it will subsequently execute as the new owner.

new_schema

The new schema for the function.

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT changes the function so that it will be invoked when some or all of its arguments are null. RETURNS NULL ON NULL INPUT or STRICT changes the function so that it is not invoked if any of its arguments are null; instead, a null result is assumed automatically. See *CREATE FUNCTION* for more information.

IMMUTABLE
STABLE
VOLATILE

Change the volatility of the function to the specified setting. See *CREATE FUNCTION* for details.

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

Change whether the function is a security definer or not. The key word EXTERNAL is ignored for SQL conformance. See CREATE FUNCTION for more information about this capability.

LEAKPROOF

Change whether the function is considered leakproof or not. See *CREATE FUNCTION* for more information about this capability.

EXECUTE ON ANY
EXECUTE ON MASTER
EXECUTE ON ALL SEGMENTS

EXECUTE ON INITPLAN

The `EXECUTE ON` attributes specify where (master or segment instance) a function executes when it is invoked during the query execution process.

`EXECUTE ON ANY` (the default) indicates that the function can be executed on the master, or any segment instance, and it returns the same result regardless of where it is executed. Greenplum Database determines where the function executes.

`EXECUTE ON MASTER` indicates that the function must execute only on the master instance.

`EXECUTE ON ALL SEGMENTS` indicates that the function must execute on all primary segment instances, but not the master, for each invocation. The overall result of the function is the `UNION ALL` of the results from all segment instances.

`EXECUTE ON INITPLAN` indicates that the function contains an SQL command that dispatches queries to the segment instances and requires special processing on the master instance by Greenplum Database when possible.

For more information about the `EXECUTE ON` attributes, see *CREATE FUNCTION*.

COST *execution_cost*

Change the estimated execution cost of the function. See *CREATE FUNCTION* for more information.

configuration_parameter
value

Set or change the value of a configuration parameter when the function is called. If *value* is `DEFAULT` or, equivalently, `RESET` is used, the function-local setting is removed, and the function executes with the value present in its environment. Use `RESET ALL` to clear all function-local settings. `SET FROM CURRENT` saves the value of the parameter that is current when `ALTER FUNCTION` is executed as the value to be applied when the function is entered.

RESTRICT

Ignored for conformance with the SQL standard.

Notes

Greenplum Database has limitations on the use of functions defined as `STABLE` or `VOLATILE`. See *CREATE FUNCTION* for more information.

Examples

To rename the function `sqrt` for type `integer` to `square_root`:

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

To change the owner of the function `sqrt` for type `integer` to `joe`:

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

To change the *schema* of the function `sqrt` for type `integer` to `math`:

```
ALTER FUNCTION sqrt(integer) SET SCHEMA math;
```

To adjust the search path that is automatically set for a function:

```
ALTER FUNCTION check_password(text) RESET search_path;
```

Compatibility

This statement is partially compatible with the `ALTER FUNCTION` statement in the SQL standard. The standard allows more properties of a function to be modified, but does not provide the ability to rename a function, make a function a security definer, or change the owner, schema, or volatility of a function. The standard also requires the `RESTRICT` key word, which is optional in Greenplum Database.

See Also

`CREATE FUNCTION`, *`DROP FUNCTION`*

ALTER GROUP

Changes a role name or membership.

Synopsis

```
ALTER GROUP groupname ADD USER username [, ... ]

ALTER GROUP groupname DROP USER username [, ... ]

ALTER GROUP groupname RENAME TO newname
```

Description

`ALTER GROUP` changes the attributes of a user group. This is an obsolete command, though still accepted for backwards compatibility, because users and groups are superseded by the more general concept of roles. See *`ALTER ROLE`* for more information.

The first two variants add users to a group or remove them from a group. Any role can play the part of *groupname* or *username*. The preferred method for accomplishing these tasks is to use *GRANT* and *REVOKE*.

Parameters

groupname

The name of the group (role) to modify.

username

Users (roles) that are to be added to or removed from the group. The users (roles) must already exist.

newname

The new name of the group (role).

Examples

To add users to a group:

```
ALTER GROUP staff ADD USER karl, john;
```

To remove a user from a group:

```
ALTER GROUP workers DROP USER beth;
```

Compatibility

There is no `ALTER GROUP` statement in the SQL standard.

See Also

ALTER ROLE, *GRANT*, *REVOKE*

ALTER INDEX

Changes the definition of an index.

Synopsis

```
ALTER INDEX [ IF EXISTS ] name RENAME TO new_name

ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name

ALTER INDEX [ IF EXISTS ] name SET ( storage_parameter = value [, ...] )

ALTER INDEX [ IF EXISTS ] name RESET ( storage_parameter  [, ...] )

ALTER INDEX ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
  SET TABLESPACE new_tablespace [ NOWAIT ]
```

Description

`ALTER INDEX` changes the definition of an existing index. There are several subforms:

- **RENAME** — Changes the name of the index. There is no effect on the stored data.
- **SET TABLESPACE** — Changes the index's tablespace to the specified tablespace and moves the data file(s) associated with the index to the new tablespace. To change the tablespace of an index, you must own the index and have `CREATE` privilege on the new tablespace. All indexes in the current database in a tablespace can be moved by using the `ALL IN TABLESPACE` form, which will lock all indexes to be moved and then move each one. This form also supports `OWNED BY`, which will only move indexes owned by the roles specified. If the `NOWAIT` option is specified then the command will fail if it is unable to acquire all of the locks required immediately. Note that system catalogs will not be moved by this command, use `ALTER DATABASE` or explicit `ALTER INDEX` invocations instead if desired. See also `CREATE TABLESPACE`.
- **IF EXISTS** — Do not throw an error if the index does not exist. A notice is issued in this case.
- **SET** — Changes the index-method-specific storage parameters for the index. The built-in index methods all accept a single parameter: *fillfactor*. The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. Index contents will not be modified immediately by this command. Use `REINDEX` to rebuild the index to get the desired effects.
- **RESET** — Resets storage parameters for the index to their defaults. The built-in index methods all accept a single parameter: *fillfactor*. As with `SET`, a `REINDEX` may be needed to update the index entirely.

Parameters

name

The name (optionally schema-qualified) of an existing index to alter.

new_name

New name for the index.

tablespace_name

The tablespace to which the index will be moved.

storage_parameter

The name of an index-method-specific storage parameter.

value

The new value for an index-method-specific storage parameter. This might be a number or a word depending on the parameter.

Notes

These operations are also possible using `ALTER TABLE`.

Changing any part of a system catalog index is not permitted.

Examples

To rename an existing index:

```
ALTER INDEX distributors RENAME TO suppliers;
```

To move an index to a different tablespace:

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

To change an index's fill factor (assuming that the index method supports it):

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

Compatibility

`ALTER INDEX` is a Greenplum Database extension.

See Also

`CREATE INDEX`, *`REINDEX`*, *`ALTER TABLE`*

ALTER LANGUAGE

Changes the name of a procedural language.

Synopsis

```
ALTER LANGUAGE name RENAME TO newname
ALTER LANGUAGE name OWNER TO new_owner
```

Description

`ALTER LANGUAGE` changes the definition of a procedural language for a specific database. Definition changes supported include renaming the language or assigning a new owner. You must be superuser or the owner of the language to use `ALTER LANGUAGE`.

Parameters

name

Name of a language.

newname

The new name of the language.

new_owner

The new owner of the language.

Compatibility

There is no ALTER LANGUAGE statement in the SQL standard.

See Also

CREATE LANGUAGE, *DROP LANGUAGE*

ALTER MATERIALIZED VIEW

Changes the definition of a materialized view.

Synopsis

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] name action [, ... ]
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME TO new_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER MATERIALIZED VIEW ALL IN TABLESPACE name [ OWNED BY role_name
 [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]

where action is one of:

    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
 MAIN }
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET ( storage_parameter = value [, ... ] )
    RESET ( storage_parameter [, ... ] )
    OWNER TO new_owner
```

Description

ALTER MATERIALIZED VIEW changes various auxiliary properties of an existing materialized view.

You must own the materialized view to use ALTER MATERIALIZED VIEW. To change a materialized view's schema, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the materialized view's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the materialized view. However, a superuser can alter ownership of any view anyway.)

The statement subforms and actions available for ALTER MATERIALIZED VIEW are a subset of those available for ALTER TABLE, and have the same meaning when used for materialized views. See the descriptions for *ALTER TABLE* for details.

Parameters

name

The name (optionally schema-qualified) of an existing materialized view.

column_name

Name of a new or existing column.

new_column_name

New name for an existing column.

new_owner

The user name of the new owner of the materialized view.

new_name

The new name for the materialized view.

new_schema

The new schema for the materialized view.

Examples

To rename the materialized view foo to bar:

```
ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

Compatibility

`ALTER MATERIALIZED VIEW` is a Greenplum Database extension of the SQL standard.

See Also

CREATE MATERIALIZED VIEW, *DROP MATERIALIZED VIEW*, *REFRESH MATERIALIZED VIEW*

ALTER OPERATOR

Changes the definition of an operator.

Synopsis

```
ALTER OPERATOR name ( {left_type | NONE} , {right_type | NONE} )
   OWNER TO new_owner

ALTER OPERATOR name ( {left_type | NONE} , {right_type | NONE} )
    SET SCHEMA new_schema
```

Description

`ALTER OPERATOR` changes the definition of an operator. The only currently available functionality is to change the owner of the operator.

You must own the operator to use `ALTER OPERATOR`. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the operator's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the operator. However, a superuser can alter ownership of any operator anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing operator.

left_type

The data type of the operator's left operand; write `NONE` if the operator has no left operand.

right_type

The data type of the operator's right operand; write `NONE` if the operator has no right operand.

new_owner

The new owner of the operator.

new_schema

The new schema for the operator.

Examples

Change the owner of a custom operator a `@@` b for type `text`:

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

Compatibility

There is no `ALTEROPERATOR` statement in the SQL standard.

See Also

`CREATE OPERATOR`, *`DROP OPERATOR`*

ALTER OPERATOR CLASS

Changes the definition of an operator class.

Synopsis

```
ALTER OPERATOR CLASS name USING index_method RENAME TO new_name

ALTER OPERATOR CLASS name USING index_method OWNER TO new_owner

ALTER OPERATOR CLASS name USING index_method SET SCHEMA new_schema
```

Description

`ALTER OPERATOR CLASS` changes the definition of an operator class.

You must own the operator class to use `ALTER OPERATOR CLASS`. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the operator class's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the operator class. However, a superuser can alter ownership of any operator class anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing operator class.

index_method

The name of the index method this operator class is for.

new_name

The new name of the operator class.

new_owner

The new owner of the operator class

new_schema

The new schema for the operator class.

Compatibility

There is no `ALTER OPERATOR CLASS` statement in the SQL standard.

See Also

`CREATE OPERATOR CLASS`, *`DROP OPERATOR CLASS`*

ALTER OPERATOR FAMILY

Changes the definition of an operator family.

Synopsis

```
ALTER OPERATOR FAMILY name USING index_method ADD
  {  OPERATOR strategy_number operator_name ( op_type, op_type ) [ FOR
 SEARCH | FOR ORDER BY sort_family_name ]
    | FUNCTION support_number [ ( op_type [ , op_type ] ) ] funcname
 ( argument_type [, ...] )
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method DROP
  {  OPERATOR strategy_number ( op_type, op_type )
    | FUNCTION support_number [ ( op_type [ , op_type ] )
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method RENAME TO new_name

ALTER OPERATOR FAMILY name USING index_method OWNER TO new_owner

ALTER OPERATOR FAMILY name USING index_method SET SCHEMA new_schema
```

Description

`ALTER OPERATOR FAMILY` changes the definition of an operator family. You can add operators and support functions to the family, remove them from the family, or change the family's name or owner.

When operators and support functions are added to a family with `ALTER OPERATOR FAMILY`, they are not part of any specific operator class within the family, but are just "loose" within the family. This indicates that these operators and functions are compatible with the family's semantics, but are not required for correct functioning of any specific index. (Operators and functions that are so required should be declared as part of an operator class, instead; see *CREATE OPERATOR CLASS*.) You can drop loose members of a family from the family at any time, but members of an operator class cannot be dropped without dropping the whole class and any indexes that depend on it. Typically, single-data-type operators and functions are part of operator classes because they are needed to support an index on that specific data type, while cross-data-type operators and functions are made loose members of the family.

You must be a superuser to use `ALTER OPERATOR FAMILY`. (This restriction is made because an erroneous operator family definition could confuse or even crash the server.)

`ALTER OPERATOR FAMILY` does not presently check whether the operator family definition includes all the operators and functions required by the index method, nor whether the operators and functions form a self-consistent set. It is the user's responsibility to define a valid operator family.

`OPERATOR` and `FUNCTION` clauses can appear in any order.

Parameters

name

The name (optionally schema-qualified) of an existing operator family.

index_method

The name of the index method this operator family is for.

strategy_number

The index method's strategy number for an operator associated with the operator family.

operator_name

The name (optionally schema-qualified) of an operator associated with the operator family.

op_type

In an `OPERATOR` clause, the operand data type(s) of the operator, or `NONE` to signify a left-unary or right-unary operator. Unlike the comparable syntax in `CREATE OPERATOR CLASS`, the operand data types must always be specified. In an `ADD FUNCTION` clause, the operand data type(s) the function is intended to support, if different from the input data type(s) of the function. For B-tree comparison functions it is not necessary to specify *op_type* since the function's input data type(s) are always the correct ones to use. For B-tree sort support functions and all functions in GiST, SP-GiST, and GIN operator classes, it is necessary to specify the operand data type(s) the function is to be used with.

sort_family_name

The name (optionally schema-qualified) of an existing `btree` operator family that describes the sort ordering associated with an ordering operator.

If neither `FOR SEARCH` nor `FOR ORDER BY` is specified, `FOR SEARCH` is the default.

support_number

The index method's support procedure number for a function associated with the operator family.

funcname

The name (optionally schema-qualified) of a function that is an index method support procedure for the operator family.

argument_types

The parameter data type(s) of the function.

new_name

The new name of the operator family.

new_owner

The new owner of the operator family.

new_schema

The new schema for the operator family.

Compatibility

There is no `ALTER OPERATOR FAMILY` statement in the SQL standard.

Notes

Notice that the `DROP` syntax only specifies the "slot" in the operator family, by strategy or support number and input data type(s). The name of the operator or function occupying the slot is not mentioned. Also, for `DROP FUNCTION` the type(s) to specify are the input data type(s) the function is intended to support; for `GiST`, `SP_GiST`, and `GIN` indexes this might have nothing to do with the actual input argument types of the function.

Because the index machinery does not check access permissions on functions before using them, including a function or operator in an operator family is tantamount to granting public execute permission on it. This is usually not an issue for the sorts of functions that are useful in an operator family.

The operators should not be defined by SQL functions. A SQL function is likely to be inlined into the calling query, which will prevent the optimizer from recognizing that the query matches an index.

Before Greenplum Database 6.0, the `OPERATOR` clause could include a `RECHECK` option. This option is no longer supported. Greenplum Database now determines whether an index operator is "lossy" on-the-fly at run time. This allows more efficient handling of cases where an operator might or might not be lossy.

Examples

The following example command adds cross-data-type operators and support functions to an operator family that already contains B-tree operator classes for data types int4 and int2.:

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

To remove these entries:

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP

  -- int4 vs int2
  OPERATOR 1 (int4, int2) ,
  OPERATOR 2 (int4, int2) ,
  OPERATOR 3 (int4, int2) ,
  OPERATOR 4 (int4, int2) ,
  OPERATOR 5 (int4, int2) ,
  FUNCTION 1 (int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 (int2, int4) ,
  OPERATOR 2 (int2, int4) ,
  OPERATOR 3 (int2, int4) ,
  OPERATOR 4 (int2, int4) ,
  OPERATOR 5 (int2, int4) ,
  FUNCTION 1 (int2, int4) ;
```

See Also

CREATE OPERATOR FAMILY, *DROP OPERATOR FAMILY*, *ALTER OPERATOR CLASS*, *CREATE OPERATOR CLASS*, *DROP OPERATOR CLASS*

ALTER PROTOCOL

Changes the definition of a protocol.

Synopsis

```
ALTER PROTOCOL name RENAME TO newname

ALTER PROTOCOL name OWNER TO newowner
```

Description

`ALTER PROTOCOL` changes the definition of a protocol. Only the protocol name or owner can be altered.

You must own the protocol to use `ALTER PROTOCOL`. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on schema of the conversion.

These restrictions are in place to ensure that altering the owner only makes changes that could by made by dropping and recreating the protocol. Note that a superuser can alter ownership of any protocol.

Parameters

name

The name (optionally schema-qualified) of an existing protocol.

newname

The new name of the protocol.

newowner

The new owner of the protocol.

Examples

To rename the conversion `GPDBauth` to `GPDB_authentication`:

```
ALTER PROTOCOL GPDBauth RENAME TO GPDB_authentication;
```

To change the owner of the conversion `GPDB_authentication` to `joe`:

```
ALTER PROTOCOL GPDB_authentication OWNER TO joe;
```

Compatibility

There is no `ALTER PROTOCOL` statement in the SQL standard.

See Also

CREATE EXTERNAL TABLE, *CREATE PROTOCOL*

ALTER RESOURCE GROUP

Changes the limits of a resource group.

Synopsis

```
ALTER RESOURCE GROUP name SET group_attribute value
```

where *group_attribute* is one of:

```
CONCURRENCY integer
CPU_RATE_LIMIT integer
CPUSET tuple
```

```
MEMORY_LIMIT integer
MEMORY_SHARED_QUOTA integer
MEMORY_SPILL_RATIO integer
```

Description

`ALTER RESOURCE GROUP` changes the limits of a resource group. Only a superuser can alter a resource group.

You can set or reset the concurrency limit of a resource group that you create for roles to control the maximum number of active concurrent statements in that group. You can also reset the memory or CPU resources of a resource group to control the amount of memory or CPU resources that all queries submitted through the group can consume on each segment host.

When you alter the CPU resource management mode or limit of a resource group, the new mode or limit is immediately applied.

When you alter a memory limit of a resource group that you create for roles, the new resource limit is immediately applied if current resource usage is less than or equal to the new value and there are no running transactions in the resource group. If the current resource usage exceeds the new memory limit value, or if there are running transactions in other resource groups that hold some of the resource, then Greenplum Database defers assigning the new limit until resource usage falls within the range of the new value.

When you increase the memory limit of a resource group that you create for external components, the new resource limit is phased in as system memory resources become available. If you decrease the memory limit of a resource group that you create for external components, the behavior is component-specific. For example, if you decrease the memory limit of a resource group that you create for a PL/Container runtime, queries in a running container may fail with an out of memory error.

You can alter one limit type in a single `ALTER RESOURCE GROUP` call.

Parameters

name

The name of the resource group to alter.

`CONCURRENCY` *`integer`*

The maximum number of concurrent transactions, including active and idle transactions, that are permitted for resource groups that you assign to roles. Any transactions submitted after the `CONCURRENCY` value limit is reached are queued. When a running transaction completes, the earliest queued transaction is executed.

The `CONCURRENCY` value must be an integer in the range [0 .. `max_connections`]. The default `CONCURRENCY` value for a resource group that you create for roles is 20.

> **Note:** You cannot set the `CONCURRENCY` value for the `admin_group` to zero (0).

`CPU_RATE_LIMIT` *`integer`*

The percentage of CPU resources to allocate to this resource group. The minimum CPU percentage for a resource group is 1. The maximum is 100. The sum of the `CPU_RATE_LIMIT`s of all resource groups defined in the Greenplum Database cluster must not exceed 100.

If you alter the `CPU_RATE_LIMIT` of a resource group in which you previously configured a `CPUSET`, `CPUSET` is disabled, the reserved CPU cores are returned to Greenplum Database, and `CPUSET` is set to -1.

`CPUSET` *`tuple`*

The CPU cores to reserve for this resource group. The CPU cores that you specify in *tuple* must be available in the system and cannot overlap with any CPU cores that you specify for other resource groups.

tuple is a comma-separated list of single core numbers or core intervals. You must enclose *tuple* in single quotes, for example, '1,3-4'.

If you alter the `CPUSET` value of a resource group for which you previously configured a `CPU_RATE_LIMIT`, `CPU_RATE_LIMIT` is disabled, the reserved CPU resources are returned to Greenplum Database, and `CPU_RATE_LIMIT` is set to -1.

You can alter `CPUSET` for a resource group only after you have enabled resource group-based resource management for your Greenplum Database cluster.

`MEMORY_LIMIT` *`integer`*

The percentage of Greenplum Database memory resources to reserve for this resource group. The minimum memory percentage for a resource group is 0. The maximum is 100. The default value is 0.

When `MEMORY_LIMIT` is 0, Greenplum Database reserves no memory for the resource group, but uses global shared memory to fulfill all memory requests in the group. If `MEMORY_LIMIT` is 0, `MEMORY_SPILL_RATIO` must also be 0.

The sum of the `MEMORY_LIMIT`s of all resource groups defined in the Greenplum Database cluster must not exceed 100. If this sum is less than 100, Greenplum Database allocates any unreserved memory to a resource group global shared memory pool.

`MEMORY_SHARED_QUOTA` *`integer`*

The percentage of memory resources to share among transactions in the resource group. The minimum memory shared quota percentage for a resource group is 0. The maximum is 100. The default `MEMORY_SHARED_QUOTA` value is 80.

`MEMORY_SPILL_RATIO` *`integer`*

The memory usage threshold for memory-intensive operators in a transaction. You can specify an integer percentage value from 0 to 100 inclusive. The default `MEMORY_SPILL_RATIO` value is 0. When `MEMORY_SPILL_RATIO` is 0, Greenplum Database uses the *`statement_mem`* server configuration parameter value to control initial query operator memory.

Notes

Use *`CREATE ROLE`* or *`ALTER ROLE`* to assign a specific resource group to a role (user).

You cannot submit an `ALTER RESOURCE GROUP` command in an explicit transaction or sub-transaction.

Examples

Change the active transaction limit for a resource group:

```
ALTER RESOURCE GROUP rgroup1 SET CONCURRENCY 13;
```

Update the CPU limit for a resource group:

```
ALTER RESOURCE GROUP rgroup2 SET CPU_RATE_LIMIT 45;
```

Update the memory limit for a resource group:

```
ALTER RESOURCE GROUP rgroup3 SET MEMORY_LIMIT 30;
```

Update the memory spill ratio for a resource group:

```
ALTER RESOURCE GROUP rgroup4 SET MEMORY_SPILL_RATIO 25;
```

Reserve CPU core 1 for a resource group:

```
ALTER RESOURCE GROUP rgroup5 SET CPUSET '1';
```

Compatibility

The `ALTER RESOURCE GROUP` statement is a Greenplum Database extension. This command does not exist in standard PostgreSQL.

See Also

`CREATE RESOURCE GROUP`, *`DROP RESOURCE GROUP`*, *`CREATE ROLE`*, *`ALTER ROLE`*

ALTER RESOURCE QUEUE

Changes the limits of a resource queue.

Synopsis

```
ALTER RESOURCE QUEUE name WITH ( queue_attribute=value [, ... ] )
```

where *queue_attribute* is:

```
   ACTIVE_STATEMENTS=integer
   MEMORY_LIMIT='memory_units'
   MAX_COST=float
   COST_OVERCOMMIT={TRUE|FALSE}
   MIN_COST=float
   PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX}
```

```
ALTER RESOURCE QUEUE name WITHOUT ( queue_attribute [, ... ] )
```

where *queue_attribute* is:

```
   ACTIVE_STATEMENTS
   MEMORY_LIMIT
   MAX_COST
   COST_OVERCOMMIT
   MIN_COST
```

> **Note:** A resource queue must have either an `ACTIVE_STATEMENTS` or a `MAX_COST` value. Do not remove both these `queue_attributes` from a resource queue.

Description

`ALTER RESOURCE QUEUE` changes the limits of a resource queue. Only a superuser can alter a resource queue. A resource queue must have either an `ACTIVE_STATEMENTS` or a `MAX_COST` value (or it can have both). You can also set or reset priority for a resource queue to control the relative share of available CPU resources used by queries associated with the queue, or memory limit of a resource queue to control the amount of memory that all queries submitted through the queue can consume on a segment host.

`ALTER RESOURCE QUEUE WITHOUT` removes the specified limits on a resource that were previously set. A resource queue must have either an `ACTIVE_STATEMENTS` or a `MAX_COST` value. Do not remove both these `queue_attributes` from a resource queue.

Parameters

name

The name of the resource queue whose limits are to be altered.

`ACTIVE_STATEMENTS` ***integer***

The number of active statements submitted from users in this resource queue allowed on the system at any one time. The value for `ACTIVE_STATEMENTS` should be an integer greater than 0. To reset `ACTIVE_STATEMENTS` to have no limit, enter a value of `-1`.

`MEMORY_LIMIT '`*`memory_units`*`'`

Sets the total memory quota for all statements submitted from users in this resource queue. Memory units can be specified in kB, MB or GB. The minimum memory quota for a resource queue is 10MB. There is no maximum; however the upper boundary at query execution time is limited by the physical memory of a segment host. The default value is no limit (`-1`).

`MAX_COST` ***float***

The total query optimizer cost of statements submitted from users in this resource queue allowed on the system at any one time. The value for `MAX_COST` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2). To reset `MAX_COST` to have no limit, enter a value of `-1.0`.

`COST_OVERCOMMIT` ***boolean***

If a resource queue is limited based on query cost, then the administrator can allow cost overcommit (`COST_OVERCOMMIT=TRUE`, the default). This means that a query that exceeds the allowed cost threshold will be allowed to run but only when the system is idle. If `COST_OVERCOMMIT=FALSE` is specified, queries that exceed the cost limit will always be rejected and never allowed to run.

`MIN_COST` ***float***

Queries with a cost under this limit will not be queued and run immediately. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for `MIN_COST` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2). To reset `MIN_COST` to have no limit, enter a value of `-1.0`.

`PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX}`

Sets the priority of queries associated with a resource queue. Queries or statements in queues with higher priority levels will receive a larger share of available CPU resources in case of contention. Queries in low-priority queues may be delayed while higher priority queries are executed.

Notes

GPORCA and the Postgres planner utilize different query costing models and may compute different costs for the same query. The Greenplum Database resource queue resource management scheme neither differentiates nor aligns costs between GPORCA and the Postgres Planner; it uses the literal cost value returned from the optimizer to throttle queries.

When resource queue-based resource management is active, use the `MEMORY_LIMIT` and `ACTIVE_STATEMENTS` limits for resource queues rather than configuring cost-based limits. Even when using GPORCA, Greenplum Database may fall back to using the Postgres Planner for certain queries, so using cost-based limits can lead to unexpected results.

Examples

Change the active query limit for a resource queue:

```
ALTER RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20);
```

Change the memory limit for a resource queue:

```
ALTER RESOURCE QUEUE myqueue WITH (MEMORY_LIMIT='2GB');
```

Reset the maximum and minimum query cost limit for a resource queue to no limit:

```
ALTER RESOURCE QUEUE myqueue WITH (MAX_COST=-1.0,
  MIN_COST= -1.0);
```

Reset the query cost limit for a resource queue to 3^{10} (or 30000000000.0) and do not allow overcommit:

```
ALTER RESOURCE QUEUE myqueue WITH (MAX_COST=3e+10,
  COST_OVERCOMMIT=FALSE);
```

Reset the priority of queries associated with a resource queue to the minimum level:

```
ALTER RESOURCE QUEUE myqueue WITH (PRIORITY=MIN);
```

Remove the `MAX_COST` and `MEMORY_LIMIT` limits from a resource queue:

```
ALTER RESOURCE QUEUE myqueue WITHOUT (MAX_COST, MEMORY_LIMIT);
```

Compatibility

The `ALTER RESOURCE QUEUE` statement is a Greenplum Database extension. This command does not exist in standard PostgreSQL.

See Also

`CREATE RESOURCE QUEUE`, *`DROP RESOURCE QUEUE`*, *`CREATE ROLE`*, *`ALTER ROLE`*

ALTER ROLE

Changes a database role (user or group).

Synopsis

```
ALTER ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

    SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | CREATEEXTTABLE | NOCREATEEXTTABLE  [ ( attribute='value' [, ...] )
     where attributes and values are:
       type='readable'|'writable'
       protocol='gpfdist'|'http'
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | REPLICATION | NOREPLICATION
  | CONNECTION LIMIT connlimit
  | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
```

```
   | VALID UNTIL 'timestamp'

ALTER ROLE name RENAME TO new_name

ALTER ROLE { name | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE { name | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter FROM CURRENT
ALTER ROLE { name | ALL } [ IN DATABASE database_name ]
 RESET configuration_parameter
ALTER ROLE { name | ALL } [ IN DATABASE database_name ] RESET ALL
ALTER ROLE name RESOURCE QUEUE {queue_name | NONE}
ALTER ROLE name RESOURCE GROUP {group_name | NONE}
```

Description

`ALTER ROLE` changes the attributes of a Greenplum Database role. There are several variants of this command.

`WITH option`

Changes many of the role attributes that can be specified in *`CREATE ROLE`*. (All of the possible attributes are covered, execept that there are no options for adding or removing memberships; use *`GRANT`* and *`REVOKE`* for that.) Attributes not mentioned in the command retain their previous settings. Database superusers can change any of these settings for any role. Roles having `CREATEROLE` privilege can change any of these settings, but only for non-superuser and non-replication roles. Ordinary roles can only change their own password.

`RENAME`

Changes the name of the role. Database superusers can rename any role. Roles having `CREATEROLE` privilege can rename non-superuser roles. The current session user cannot be renamed (connect as a different user to rename a role). Because MD5-encrypted passwords use the role name as cryptographic salt, renaming a role clears its password if the password is MD5-encrypted.

`SET | RESET`

Changes a role's session default for a specified configuration parameter, either for all databases or, when the `IN DATABASE` clause is specified, only for sessions in the named database. If `ALL` is specified instead of a role name, this changes the setting for all roles. Using `ALL` with `IN DATABASE` is effectively the same as using the command `ALTER DATABASE...SET...`.

Whenever the role subsequently starts a new session, the specified value becomes the session default, overriding whatever setting is present in the server configuration file (`postgresql.conf`) or has been received from the `postgres` command line. This only happens at login time; executing *`SET ROLE`* or *`SET SESSION AUTHORIZATION`* does not cause new configuration values to be set.

Database-specific settings attached to a role override settings for all databases. Settings for specific databases or specific roles override settings for all roles.

For a role without `LOGIN` privilege, session defaults have no effect. Ordinary roles can change their own session defaults. Superusers can change anyone's session defaults. Roles having `CREATEROLE` privilege can change defaults for non-superuser roles. Ordinary roles can only set defaults for themselves. Certain configuration variables cannot be set this way, or can only be set if a superuser issues the command. See the *Greenplum Database Reference Guide* for information about all user-settable configuration parameters. Only superusers can change a setting for all roles in all databases.

`RESOURCE QUEUE`

Assigns the role to a resource queue. The role would then be subject to the limits assigned to the resource queue when issuing queries. Specify `NONE` to assign the role to the default resource queue. A role can only belong to one resource queue. For a role without `LOGIN` privilege, resource queues have no effect. See *`CREATE RESOURCE QUEUE`* for more information.

`RESOURCE GROUP`

Assigns a resource group to the role. The role would then be subject to the concurrent transaction, memory, and CPU limits configured for the resource group. You can assign a single resource group to one or more roles. You cannot assign a resource group that you create for an external component to a role. See *`CREATE RESOURCE GROUP`* for additional information.

Parameters

`name`

The name of the role whose attributes are to be altered.

`new_name`

The new name of the role.

`database_name`

The name of the database in which to set the configuration parameter.

`config_parameter=value`

Set this role's session default for the specified configuration parameter to the given value. If *value* is `DEFAULT` or if `RESET` is used, the role-specific parameter setting is removed, so the role will inherit the system-wide default setting in new sessions. Use `RESET ALL` to clear all role-specific settings. `SET FROM CURRENT` saves the session's current value of the parameter as the role-specific value. If `IN DATABASE` is specified, the configuration parameter is set or removed for the given role and database only. Whenever the role subsequently starts a new session, the specified value becomes the session default, overriding whatever setting is present in `postgresql.conf` or has been received from the `postgres` command line.

Role-specific variable settings take effect only at login; `SET ROLE` and *`SET SESSION AUTHORIZATION`* do not process role-specific variable settings.

See *Server Configuration Parameters* for information about user-settable configuration parameters.

`group_name`

The name of the resource group to assign to this role. Specifying the *group_name* `NONE` removes the role's current resource group assignment and assigns a default resource group based on the role's capability. `SUPERUSER` roles are assigned the `admin_group` resource group, while the `default_group` resource group is assigned to non-admin roles.

You cannot assign a resource group that you create for an external component to a role.

`queue_name`

The name of the resource queue to which the user-level role is to be assigned. Only roles with `LOGIN` privilege can be assigned to a resource queue. To unassign a role from a resource queue and put it in the default resource queue, specify `NONE`. A role can only belong to one resource queue.

`SUPERUSER | NOSUPERUSER`
`CREATEDB | NOCREATEDB`
`CREATEROLE | NOCREATEROLE`
`CREATEUSER | NOCREATEUSER`

CREATEUSER and NOCREATEUSER are obsolete, but still accepted, spellings of SUPERUSER and NOSUPERUSER. Note that they are not equivalent to the CREATEROLE and NOCREATEROLE clauses.

CREATEEXTTABLE | NOCREATEEXTTABLE [(attribute='value')]

If CREATEEXTTABLE is specified, the role being defined is allowed to create external tables. The default `type` is `readable` and the default `protocol` is `gpfdist` if not specified. NOCREATEEXTTABLE (the default) denies the role the ability to create external tables. Note that external tables that use the `file` or `execute` protocols can only be created by superusers.

INHERIT | NOINHERIT
LOGIN | NOLOGIN
REPLICATION
NOREPLICATION
CONNECTION LIMIT connlimit
PASSWORD password
ENCRYPTED | UNENCRYPTED
VALID UNTIL 'timestamp'

These clauses alter role attributes originally set by *CREATE ROLE*.

DENY deny_point
DENY BETWEEN deny_point AND deny_point

The DENY and DENY BETWEEN keywords set time-based constraints that are enforced at login. DENYsets a day or a day and time to deny access. DENY BETWEEN sets an interval during which access is denied. Both use the parameter *deny_point* that has following format:

```
DAY day [ TIME 'time' ]
```

The two parts of the `deny_point` parameter use the following formats:

For day:

```
{'Sunday' | 'Monday' | 'Tuesday' |'Wednesday' | 'Thursday' |
 'Friday' |
'Saturday' | 0-6 }
```

For `time:`

{ 00-23 : 00-59 | 01-12 : 00-59 { AM | PM }}

The DENY BETWEEN clause uses two *deny_point* parameters.

```
DENY BETWEEN deny_point AND deny_point
```

For more information about time-based constraints and examples, see "Managing Roles and Privileges" in the *Greenplum Database Administrator Guide*.

DROP DENY FOR deny_point

The DROP DENY FOR clause removes a time-based constraint from the role. It uses the *deny_point* parameter described above.

For more information about time-based constraints and examples, see "Managing Roles and Privileges" in the *Greenplum Database Administrator Guide*.

Notes

Use *CREATE ROLE* to add new roles, and *DROP ROLE* to remove a role.

Use *GRANT* and *REVOKE* for adding and removing role memberships.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in clear text, and it might also be logged in the client's command history or the server log. The `psql` command-line client contains a meta-command `\password` that can be used to change a role's password without exposing the clear text password.

It is also possible to tie a session default to a specific database rather than to a role; see *ALTER DATABASE*. If there is a conflict, database-role-specific settings override role-specific ones, which in turn override database-specific ones.

Examples

Change the password for a role:

```
ALTER ROLE daria WITH PASSWORD 'passwd123';
```

Remove a role's password:

```
ALTER ROLE daria WITH PASSWORD NULL;
```

Change a password expiration date:

```
ALTER ROLE scott VALID UNTIL 'May 4 12:00:00 2015 +1';
```

Make a password valid forever:

```
ALTER ROLE luke VALID UNTIL 'infinity';
```

Give a role the ability to create other roles and new databases:

```
ALTER ROLE joelle CREATEROLE CREATEDB;
```

Give a role a non-default setting of the `maintenance_work_mem` parameter:

```
ALTER ROLE admin SET maintenance_work_mem = 100000;
```

Give a role a non-default, database-specific setting of the `client_min_messages` parameter:

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```

Assign a role to a resource queue:

```
ALTER ROLE sammy RESOURCE QUEUE poweruser;
```

Give a role permission to create writable external tables:

```
ALTER ROLE load CREATEEXTTABLE (type='writable');
```

Alter a role so it does not allow login access on Sundays:

```
ALTER ROLE user3 DENY DAY 'Sunday';
```

Alter a role to remove the constraint that does not allow login access on Sundays:

```
ALTER ROLE user3 DROP DENY FOR DAY 'Sunday';
```

Assign a new resource group to a role:

```
ALTER ROLE parttime_user RESOURCE GROUP rg_light;
```

Compatibility

The ALTER ROLE statement is a Greenplum Database extension.

See Also

CREATE ROLE, *DROP ROLE*, *ALTER DATABASE*, *SET*, *CREATE RESOURCE GROUP*, *CREATE RESOURCE QUEUE*, *GRANT*, *REVOKE*

ALTER SCHEMA

Changes the definition of a schema.

Synopsis

```
ALTER SCHEMA name RENAME TO newname

ALTER SCHEMA name OWNER TO newowner
```

Description

ALTER SCHEMA changes the definition of a schema.

You must own the schema to use ALTER SCHEMA. To rename a schema you must also have the CREATE privilege for the database. To alter the owner, you must also be a direct or indirect member of the new owning role, and you must have the CREATE privilege for the database. Note that superusers have all these privileges automatically.

Parameters

name

The name of an existing schema.

newname

The new name of the schema. The new name cannot begin with pg_, as such names are reserved for system schemas.

newowner

The new owner of the schema.

Compatibility

There is no ALTER SCHEMA statement in the SQL standard.

See Also

CREATE SCHEMA, *DROP SCHEMA*

ALTER SEQUENCE

Changes the definition of a sequence generator.

Synopsis

```
ALTER SEQUENCE [ IF EXISTS ] name [INCREMENT [ BY ] increment]
     [MINVALUE minvalue | NO MINVALUE]
     [MAXVALUE maxvalue | NO MAXVALUE]
     [START [ WITH ] start ]
     [RESTART [ [ WITH ] restart] ]
```

```
      [CACHE cache] [[ NO ] CYCLE]
      [OWNED BY {table.column | NONE}]

ALTER SEQUENCE [ IF EXISTS ] name OWNER TO new_owner

ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name

ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema
```

Description

`ALTER SEQUENCE` changes the parameters of an existing sequence generator. Any parameters not specifically set in the `ALTER SEQUENCE` command retain their prior settings.

You must own the sequence to use `ALTER SEQUENCE`. To change a sequence's schema, you must also have `CREATE` privilege on the new schema. Note that superusers have all these privileges automatically.

To alter the owner, you must be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the sequence's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the sequence. However, a superuser can alter ownership of any sequence anyway.)

Parameters

name

The name (optionally schema-qualified) of a sequence to be altered.

IF EXISTS

Do not throw an error if the sequence does not exist. A notice is issued in this case.

increment

The clause `INCREMENT BY` *`increment`* is optional. A positive value will make an ascending sequence, a negative one a descending sequence. If unspecified, the old increment value will be maintained.

minvalue
NO MINVALUE

The optional clause `MINVALUE` *`minvalue`* determines the minimum value a sequence can generate. If `NO MINVALUE` is specified, the defaults of 1 and -263-1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current minimum value will be maintained.

maxvalue
NO MAXVALUE

The optional clause `MAXVALUE` *`maxvalue`* determines the maximum value for the sequence. If `NO MAXVALUE` is specified, the defaults are 263-1 and -1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current maximum value will be maintained.

start

The optional clause START WITH *start* changes the recorded start value of the sequence. This has no effect on the *current* sequence value; it simply sets the value that future `ALTER SEQUENCE RESTART` commands will use.

restart

The optional clause `RESTART [ WITH` *`restart`* `]` changes the current value of the sequence. This is equivalent to calling the `setval(`*`sequence, start_val, is_called`*`)` function with *`is_called`* `= false`. The specified value will be returned by the next call of the `nextval(`*`sequence`*`)` function. Writing `RESTART` with no *restart* value

is equivalent to supplying the start value that was recorded by `CREATE SEQUENCE` or last set by `ALTER SEQUENCE START WITH`.

new_owner

The user name of the new owner of the sequence.

cache

The clause `CACHE` *`cache`* enables sequence numbers to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache). If unspecified, the old cache value will be maintained.

CYCLE

The optional `CYCLE` key word may be used to enable the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence. If the limit is reached, the next number generated will be the respective *minvalue* or *maxvalue*.

NO CYCLE

If the optional `NO CYCLE` key word is specified, any calls to `nextval()` after the sequence has reached its maximum value will return an error. If neither `CYCLE` or `NO CYCLE` are specified, the old cycle behavior will be maintained.

OWNED BY *table.column*
OWNED BY NONE

The `OWNED BY` option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. If specified, this association replaces any previously specified association for the sequence. The specified table must have the same owner and be in the same schema as the sequence. Specifying `OWNED BY NONE` removes any existing table column association.

new_name

The new name for the sequence.

new_schema

The new schema for the sequence.

Notes

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, `ALTER SEQUENCE`'s effects on the sequence generation parameters are never rolled back; those changes take effect immediately and are not reversible. However, the `OWNED BY`, `OWNER TO`, `RENAME TO`, and `SET SCHEMA` clauses are ordinary catalog updates and can be rolled back.

`ALTER SEQUENCE` will not immediately affect `nextval()` results in sessions, other than the current one, that have preallocated (cached) sequence values. They will use up all cached values prior to noticing the changed sequence generation parameters. The current session will be affected immediately.

For historical reasons, `ALTER TABLE` can be used with sequences too; but the only variants of `ALTER TABLE` that are allowed with sequences are equivalent to the forms shown above.

Examples

Restart a sequence called `serial` at `105`:

```
ALTER SEQUENCE serial RESTART WITH 105;
```

Compatibility

`ALTER SEQUENCE` conforms to the SQL standard, except for the `START WITH`, `OWNED BY`, `OWNER TO`, `RENAME TO`, and `SET SCHEMA` clauses, which are Greenplum Database extensions.

See Also

`CREATE SEQUENCE`, *`DROP SEQUENCE`*, *`ALTER TABLE`*

ALTER SERVER

Changes the definition of a foreign server.

Synopsis

```
ALTER SERVER server_name [ VERSION 'new_version' ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]

ALTER SERVER server_name OWNER TO new_owner

ALTER SERVER server_name RENAME TO new_name
```

Description

`ALTER SERVER` changes the definition of a foreign server. The first form of the command changes the version string or the generic options of the server. Greenplum Database requires at least one clause. The second and third forms of the command change the owner or the name of the server.

To alter the server, you must be the owner of the server. To alter the owner you must:

- Own the server.
- Be a direct or indirect member of the new owning role.
- Have `USAGE` privilege on the server's foreign-data wrapper.

Superusers automatically satisfy all of these criteria.

Parameters

server_name

The name of an existing server.

new_version

The new server version.

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

Change the server's options. `ADD`, `SET`, and `DROP` specify the action to perform. If no operation is explicitly specified, the default operation is `ADD`. Option names must be unique. Greenplum Database validates names and values using the server's foreign-data wrapper library.

OWNER TO *new_owner*

Specifies the new owner of the foreign server.

RENAME TO *new_name*

Specifies the new name of the foreign server.

Examples

Change the definition of a server named `foo` by adding connection options:

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

Change the option named `host` for a server named `foo`, and set the server version:

```
ALTER SERVER foo VERSION '9.1' OPTIONS (SET host 'baz');
```

Compatibility

`ALTER SERVER` conforms to ISO/IEC 9075-9 (SQL/MED). The `OWNER TO` and `RENAME` forms are Greenplum Database extensions.

See Also

`CREATE SERVER`, *`DROP SERVER`*

ALTER TABLE

Changes the definition of a table.

Synopsis

```
ALTER TABLE [IF EXISTS] [ONLY] name
    action [, ... ]

ALTER TABLE [IF EXISTS] [ONLY] name
    RENAME [COLUMN] column_name TO new_column_name

ALTER TABLE [ IF EXISTS ] [ ONLY ] name
    RENAME CONSTRAINT constraint_name TO new_constraint_name

ALTER TABLE [IF EXISTS] name
    RENAME TO new_name

ALTER TABLE [IF EXISTS] name
    SET SCHEMA new_schema

ALTER TABLE ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]

ALTER TABLE [IF EXISTS] [ONLY] name SET
     WITH (REORGANIZE=true|false)
   | DISTRIBUTED BY ({column_name [opclass]} [, ... ] )
   | DISTRIBUTED RANDOMLY
   | DISTRIBUTED REPLICATED

ALTER TABLE name
   [ ALTER PARTITION { partition_name | FOR (RANK(number))
   | FOR (value) } [...] ] partition_action

where action is one of:

  ADD [COLUMN] column_name data_type [ DEFAULT default_expr ]
      [column_constraint [ ... ]]
      [ COLLATE collation ]
      [ ENCODING ( storage_directive [,...] ) ]
  DROP [COLUMN] [IF EXISTS] column_name [RESTRICT | CASCADE]
```

```
  ALTER [COLUMN] column_name [ SET DATA ] TYPE type [COLLATE collation]
 [USING expression]
  ALTER [COLUMN] column_name SET DEFAULT expression
  ALTER [COLUMN] column_name DROP DEFAULT
  ALTER [COLUMN] column_name { SET | DROP } NOT NULL
  ALTER [COLUMN] column_name SET STATISTICS integer
  ALTER [COLUMN] column SET ( attribute_option = value [, ... ] )
  ALTER [COLUMN] column RESET ( attribute_option [, ... ] )
  ADD table_constraint [NOT VALID]
  ADD table_constraint_using_index
  VALIDATE CONSTRAINT constraint_name
  DROP CONSTRAINT [IF EXISTS] constraint_name [RESTRICT | CASCADE]
  DISABLE TRIGGER [trigger_name | ALL | USER]
  ENABLE TRIGGER [trigger_name | ALL | USER]
  CLUSTER ON index_name
  SET WITHOUT CLUSTER
  SET WITHOUT OIDS
  SET (storage_parameter = value)
  RESET (storage_parameter [, ... ])
  INHERIT parent_table
  NO INHERIT parent_table
  OF type_name
  NOT OF
  OWNER TO new_owner
  SET TABLESPACE new_tablespace
```

where *table_constraint_using_index* is:

```
 [ CONSTRAINT constraint_name ]
 { UNIQUE | PRIMARY KEY } USING INDEX index_name
 [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

where *partition_action* is one of:

```
  ALTER DEFAULT PARTITION
  DROP DEFAULT PARTITION [IF EXISTS]
  DROP PARTITION [IF EXISTS] { partition_name |
      FOR (RANK(number)) | FOR (value) } [CASCADE]
  TRUNCATE DEFAULT PARTITION
  TRUNCATE PARTITION { partition_name | FOR (RANK(number)) |
      FOR (value) }
  RENAME DEFAULT PARTITION TO new_partition_name
  RENAME PARTITION { partition_name | FOR (RANK(number)) |
      FOR (value) } TO new_partition_name
  ADD DEFAULT PARTITION name [ ( subpartition_spec ) ]
  ADD PARTITION [partition_name] partition_element
     [ ( subpartition_spec ) ]
  EXCHANGE PARTITION { partition_name | FOR (RANK(number)) |
       FOR (value) } WITH TABLE table_name
        [ WITH | WITHOUT VALIDATION ]
  EXCHANGE DEFAULT PARTITION WITH TABLE table_name
   [ WITH | WITHOUT VALIDATION ]
  SET SUBPARTITION TEMPLATE (subpartition_spec)
  SPLIT DEFAULT PARTITION
    {  AT (list_value)
     | START([datatype] range_value) [INCLUSIVE | EXCLUSIVE]
       END([datatype] range_value) [INCLUSIVE | EXCLUSIVE] }
    [ INTO ( PARTITION new_partition_name,
             PARTITION default_partition_name ) ]
  SPLIT PARTITION { partition_name | FOR (RANK(number)) |
     FOR (value) } AT (value)
    [ INTO (PARTITION partition_name, PARTITION partition_name)]
```

where *partition_element* is:

```
    VALUES (list_value [,...] )
  | START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
    [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
  | END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]
```

where *subpartition_spec* is:

```
subpartition_element [, ...]
```

and *subpartition_element* is:

```
   DEFAULT SUBPARTITION subpartition_name
  | [SUBPARTITION subpartition_name] VALUES (list_value [,...] )
  | [SUBPARTITION subpartition_name]
    START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
    [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
    [ EVERY ( [number | datatype] 'interval_value') ]
  | [SUBPARTITION subpartition_name]
    END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
    [ EVERY ( [number | datatype] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]
```

where *storage_parameter* is:

```
   appendoptimized={TRUE|FALSE}
   blocksize={8192-2097152}
   orientation={COLUMN|ROW}
   compresstype={ZLIB|ZSTD|QUICKLZ|RLE_TYPE|NONE}
   compresslevel={0-9}
   fillfactor={10-100}
   [oids=FALSE]
```

Description

`ALTER TABLE` changes the definition of an existing table. There are several subforms:

- **ADD COLUMN** — Adds a new column to the table, using the same syntax as *`CREATE TABLE`*. The `ENCODING` clause is valid only for append-optimized, column-oriented tables.

  When you add a column to an append-optimized, column-oriented table, Greenplum Database sets each data compression parameter for the column (`compresstype`, `compresslevel`, and `blocksize`) based on the following setting, in order of preference.

  1. The compression parameter setting specified in the `ALTER TABLE` command `ENCODING` clause.
  2. If the server configuration parameter *`gp_add_column_inherits_table_setting`* is `on`, use the table's data compression parameters specified in the `WITH` clause when the table was created. The default server configuration parameter default is `off`, the `WITH` clause parameters are ignored.
  3. The compression parameter setting specified in the server configuration parameter *`gp_default_storage_option`*.
  4. The default compression parameter value.

  For append-optimized and hash tables, `ADD COLUMN` requires a table rewrite. For information about table rewrites performed by `ALTER TABLE`, see *Notes*.
- **DROP COLUMN [IF EXISTS]** — Drops a column from a table. Note that if you drop table columns that are being used as the Greenplum Database distribution key, the distribution policy for the table

will be changed to `DISTRIBUTED RANDOMLY`. Indexes and table constraints involving the column are automatically dropped as well. You need to say `CASCADE` if anything outside the table depends on the column (such as views). If `IF EXISTS` is specified and the column does not exist, no error is thrown; a notice is issued instead.

- **IF EXISTS** — Do not throw an error if the table does not exist. A notice is issued in this case.
- **SET DATA TYPE** — This form changes the data type of a column of a table. Note that you cannot alter column data types that are being used as distribution or partitioning keys. Indexes and simple table constraints involving the column will be automatically converted to use the new column type by reparsing the originally supplied expression. The optional `COLLATE` clause specifies a collation for the new column; if omitted, the collation is the default for the new column type. The optional `USING` clause specifies how to compute the new column value from the old. If omitted, the default conversion is the same as an assignment cast from old data type to new. A `USING` clause must be provided if there is no implicit or assignment cast from old to new type.

  Note: GPORCA supports collation only when all columns in the query use the same collation. If columns in the query use different collations, then Greenplum uses the Postgres Planner.

  Changing a column data type requires a table rewrite. For information about table rewrites performed by `ALTER TABLE`, see *Notes*.
- **SET/DROP DEFAULT** — Sets or removes the default value for a column. Default values only apply in subsequent `INSERT` or `UPDATE` commands; they do not cause rows already in the table to change.
- **SET/DROP NOT NULL** — Changes whether a column is marked to allow null values or to reject null values. You can only use `SET NOT NULL` when the column contains no null values.
- **SET STATISTICS** — Sets the per-column statistics-gathering target for subsequent `ANALYZE` operations. The target can be set in the range 100 to 10000, or set to -1 to revert to using the system default statistics target (`default_statistics_target`).
- **SET (*attribute_option* = *value* [, ...])**

  RESET (*attribute_option [, ...])— Sets or resets per-attribute options. Currently, the only defined per-attribute options are `n_distinct` and `n_distinct_inherited`, which override the number-of-distinct-values estimates made by subsequent *ANALYZE* operations. `n_distinct` affects the statistics for the table itself, while `n_distinct_inherited` affects the statistics gathered for the table plus its inheritance children. When set to a positive value, `ANALYZE` will assume that the column contains exactly the specified number of distinct non-null values. When set to a negative value, which must be greater than or equal to -1, `ANALYZE` will assume that the number of distinct non-null values in the column is linear in the size of the table; the exact count is to be computed by multiplying the estimated table size by the absolute value of the given number. For example, a value of -1 implies that all values in the column are distinct, while a value of -0.5 implies that each value appears twice on the average. This can be useful when the size of the table changes over time, since the multiplication by the number of rows in the table is not performed until query planning time. Specify a value of 0 to revert to estimating the number of distinct values normally.
- **ADD *table_constraint* [NOT VALID]** — Adds a new constraint to a table (not just a partition) using the same syntax as `CREATE TABLE`. The `NOT VALID` option is currently only allowed for foreign key and `CHECK` constraints. If the constraint is marked `NOT VALID`, Greenplum Database skips the potentially-lengthy initial check to verify that all rows in the table satisfy the constraint. The constraint will still be enforced against subsequent inserts or updates (that is, they'll fail unless there is a matching row in the referenced table, in the case of foreign keys; and they'll fail unless the new row matches the specified check constraints). But the database will not assume that the constraint holds for all rows in the table, until it is validated by using the `VALIDATE CONSTRAINT` option. Constraint checks are skipped at create table time, so the *CREATE TABLE* syntax does not include this option.
- **VALIDATE CONSTRAINT** — This form validates a foreign key constraint that was previously created as `NOT VALID`, by scanning the table to ensure there are no rows for which the constraint is not satisfied. Nothing happens if the constraint is already marked valid. The advantage of separating validation from initial creation of the constraint is that validation requires a lesser lock on the table than constraint creation does.

- **ADD *table_constraint_using_index*** — Adds a new `PRIMARY KEY` or `UNIQUE` constraint to a table based on an existing unique index. All the columns of the index will be included in the constraint. The index cannot have expression columns nor be a partial index. Also, it must be a b-tree index with default sort ordering. These restrictions ensure that the index is equivalent to one that would be built by a regular `ADD PRIMARY KEY` or `ADD UNIQUE` command.

  Adding a `PRIMARY KEY` or `UNIQUE` constraint to a table based on an existing unique index is not supported on a partitioned table.

  If `PRIMARY KEY` is specified, and the index's columns are not already marked `NOT NULL`, then this command will attempt to do `ALTER COLUMN SET NOT NULL` against each such column. That requires a full table scan to verify the column(s) contain no nulls. In all other cases, this is a fast operation.

  If a constraint name is provided then the index will be renamed to match the constraint name. Otherwise the constraint will be named the same as the index.

  After this command is executed, the index is "owned" by the constraint, in the same way as if the index had been built by a regular `ADD PRIMARY KEY` or `ADD UNIQUE` command. In particular, dropping the constraint will make the index disappear too.
- **DROP CONSTRAINT [IF EXISTS]** — Drops the specified constraint on a table. If `IF EXISTS` is specified and the constraint does not exist, no error is thrown. In this case a notice is issued instead.
- **DISABLE/ENABLE TRIGGER** — Disables or enables trigger(s) belonging to the table. A disabled trigger is still known to the system, but is not executed when its triggering event occurs. For a deferred trigger, the enable status is checked when the event occurs, not when the trigger function is actually executed. One may disable or enable a single trigger specified by name, or all triggers on the table, or only user-created triggers. Disabling or enabling constraint triggers requires superuser privileges.

  Note: triggers are not supported in Greenplum Database. Triggers in general have very limited functionality due to the parallelism of Greenplum Database.
- **CLUSTER ON/SET WITHOUT CLUSTER** — Selects or removes the default index for future `CLUSTER` operations. It does not actually re-cluster the table. Note that `CLUSTER` is not the recommended way to physically reorder a table in Greenplum Database because it takes so long. It is better to recreate the table with *`CREATE TABLE AS`* and order it by the index column(s).

  Note: `CLUSTER ON` is not supported on append-optimized tables.
- **SET WITHOUT OIDS** — Removes the OID system column from the table.

  Warning: Pivotal does not support using `SET WITH OIDS` or `oids=TRUE` to assign an OID system column. On large tables, such as those in a typical Greenplum Database system, using OIDs for table rows can cause wrap-around of the 32-bit OID counter. Once the counter wraps around, OIDs can no longer be assumed to be unique, which not only makes them useless to user applications, but can also cause problems in the Greenplum Database system catalog tables. In addition, excluding OIDs from a table reduces the space required to store the table on disk by 4 bytes per row, slightly improving performance. You cannot create OIDS on a partitioned or column-oriented table (an error is displayed). This syntax is deprecated and will be removed in a future Greenplum release.
- **SET (FILLFACTOR = *value*) / RESET (FILLFACTOR)** — Changes the fillfactor for the table. The fillfactor for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller fillfactor is specified, `INSERT` operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives `UPDATE` a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For a table whose entries are never updated, complete packing is the best choice, but in heavily updated tables smaller fillfactors are appropriate. Note that the table contents will not be modified immediately by this command. You will need to rewrite the table to get the desired effects. That can be done with *`VACUUM`* or one of the forms of `ALTER TABLE` that forces a table rewrite. For information about the forms of `ALTER TABLE` that perform a table rewrite, see *Notes*.
- **SET DISTRIBUTED** — Changes the distribution policy of a table. Changing a hash distribution policy, or changing to or from a replicated policy, will cause the table data to be physically redistributed on disk, which can be resource intensive.

- **INHERIT *parent_table* / NO INHERIT *parent_table*** — Adds or removes the target table as a child of the specified parent table. Queries against the parent will include records of its child table. To be added as a child, the target table must already contain all the same columns as the parent (it could have additional columns, too). The columns must have matching data types, and if they have `NOT NULL` constraints in the parent then they must also have `NOT NULL` constraints in the child. There must also be matching child-table constraints for all `CHECK` constraints of the parent, except those marked non-inheritable (that is, created with `ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT`) in the parent, which are ignored; all child-table constraints matched must not be marked non-inheritable. Currently `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` constraints are not considered, but this may change in the future.
- OF *type_name* — This form links the table to a composite type as though `CREATE TABLE OF` had formed it. The table's list of column names and types must precisely match that of the composite type; the presence of an `oid` system column is permitted to differ. The table must not inherit from any other table. These restrictions ensure that `CREATE TABLE OF` would permit an equivalent table definition.
- **NOT OF** — This form dissociates a typed table from its type.
- **OWNER** — Changes the owner of the table, sequence, or view to the specified user.
- **SET TABLESPACE** — Changes the table's tablespace to the specified tablespace and moves the data file(s) associated with the table to the new tablespace. Indexes on the table, if any, are not moved; but they can be moved separately with additional `SET TABLESPACE` commands. All tables in the current database in a tablespace can be moved by using the `ALL IN TABLESPACE` form, which will lock all tables to be moved first and then move each one. This form also supports `OWNED BY`, which will only move tables owned by the roles specified. If the `NOWAIT` option is specified then the command will fail if it is unable to acquire all of the locks required immediately. Note that system catalogs are not moved by this command, use `ALTER DATABASE` or explicit `ALTER TABLE` invocations instead if desired. The `information_schema` relations are not considered part of the system catalogs and will be moved. See also `CREATE TABLESPACE`. If changing the tablespace of a partitioned table, all child table partitions will also be moved to the new tablespace.
- **RENAME** — Changes the name of a table (or an index, sequence, view, or materialized view), the name of an individual column in a table, or the name of a constraint of the table. There is no effect on the stored data. Note that Greenplum Database distribution key columns cannot be renamed.
- **SET SCHEMA** — Moves the table into another schema. Associated indexes, constraints, and sequences owned by table columns are moved as well.
- **ALTER PARTITION | DROP PARTITION | RENAME PARTITION | TRUNCATE PARTITION | ADD PARTITION | SPLIT PARTITION | EXCHANGE PARTITION | SET SUBPARTITION TEMPLATE** — Changes the structure of a partitioned table. In most cases, you must go through the parent table to alter one of its child table partitions.

    Note: If you add a partition to a table that has subpartition encodings, the new partition inherits the storage directives for the subpartitions. For more information about the precedence of compression settings, see *Using Compression*.

All the forms of `ALTER TABLE` that act on a single table, except `RENAME` and `SET SCHEMA`, can be combined into a list of multiple alterations to apply together. For example, it is possible to add several columns and/or alter the type of several columns in a single command. This is particularly useful with large tables, since only one pass over the table need be made.

You must own the table to use `ALTER TABLE`. To change the schema or tablespace of a table, you must also have `CREATE` privilege on the new schema or tablespace. To add the table as a new child of a parent table, you must own the parent table as well. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the table's schema. To add a column or alter a column type or use the `OF` clause, you must also have `USAGE` privilege on the data type. A superuser has these privileges automatically.

Note: Memory usage increases significantly when a table has many partitions, if a table has compression, or if the blocksize for a table is large. If the number of relations associated with the table is large, this condition can force an operation on the table to use more memory. For example, if the table is a CO table and has a large number of columns, each column is a relation.

An operation like `ALTER TABLE ALTER COLUMN` opens all the columns in the table allocates associated buffers. If a CO table has 40 columns and 100 partitions, and the columns are compressed and the blocksize is 2 MB (with a system factor of 3), the system attempts to allocate 24 GB, that is (40 ×100) × (2 ×3) MB or 24 GB.

Parameters

`ONLY`

Only perform the operation on the table name specified. If the `ONLY` keyword is not used, the operation will be performed on the named table and any child table partitions associated with that table.

> **Note:** Adding or dropping a column, or changing a column's type, in a parent or descendant table only is not permitted. The parent table and its descendents must always have the same columns and types.

`name`

The name (possibly schema-qualified) of an existing table to alter. If `ONLY` is specified, only that table is altered. If `ONLY` is not specified, the table and all its descendant tables (if any) are updated.

> **Note:** Constraints can only be added to an entire table, not to a partition. Because of that restriction, the *name* parameter can only contain a table name, not a partition name.

`column_name`

Name of a new or existing column. Note that Greenplum Database distribution key columns must be treated with special care. Altering or dropping these columns can change the distribution policy for the table.

`new_column_name`

New name for an existing column.

`new_name`

New name for the table.

`type`

Data type of the new column, or new data type for an existing column. If changing the data type of a Greenplum distribution key column, you are only allowed to change it to a compatible type (for example, `text` to `varchar` is OK, but `text` to `int` is not).

`table_constraint`

New table constraint for the table. Note that foreign key constraints are currently not supported in Greenplum Database. Also a table is only allowed one unique constraint and the uniqueness must be within the Greenplum Database distribution key.

`constraint_name`

Name of an existing constraint to drop.

`CASCADE`

Automatically drop objects that depend on the dropped column or constraint (for example, views referencing the column).

`RESTRICT`

Refuse to drop the column or constraint if there are any dependent objects. This is the default behavior.

`trigger_name`

Name of a single trigger to disable or enable. Note that Greenplum Database does not support triggers.

ALL

Disable or enable all triggers belonging to the table including constraint related triggers. This requires superuser privilege if any of the triggers are internally generated constraint triggers such as those that are used to implement foreign key constraints or deferrable uniqueness and exclusion constraints.

USER

Disable or enable all triggers belonging to the table except for internally generated constraint triggers such as those that are used to implement foreign key constraints or deferrable uniqueness and exclusion constraints.

index_name

The index name on which the table should be marked for clustering. Note that `CLUSTER` is not the recommended way to physically reorder a table in Greenplum Database because it takes so long. It is better to recreate the table with *`CREATE TABLE AS`* and order it by the index column(s).

FILLFACTOR

Set the fillfactor percentage for a table.

value

The new value for the `FILLFACTOR` parameter, which is a percentage between 10 and 100. 100 is the default.

DISTRIBUTED BY ({*column_name* [*opclass*]}) | DISTRIBUTED RANDOMLY | DISTRIBUTED REPLICATED

Specifies the distribution policy for a table. Changing a hash distribution policy causes the table data to be physically redistributed, which can be resource intensive. If you declare the same hash distribution policy or change from hash to random distribution, data will not be redistributed unless you declare `SET WITH (REORGANIZE=true)`.

Changing to or from a replicated distribution policy causes the table data to be redistributed.

REORGANIZE=true|false

Use `REORGANIZE=true` when the hash distribution policy has not changed or when you have changed from a hash to a random distribution, and you want to redistribute the data anyways.

parent_table

A parent table to associate or de-associate with this table.

new_owner

The role name of the new owner of the table.

new_tablespace

The name of the tablespace to which the table will be moved.

new_schema

The name of the schema to which the table will be moved.

parent_table_name

When altering a partitioned table, the name of the top-level parent table.

ALTER [DEFAULT] PARTITION

If altering a partition deeper than the first level of partitions, use `ALTER PARTITION` clauses to specify which subpartition in the hierarchy you want to alter. For each partition level in the table hierarchy that is above the target partition, specify the partition that is related to the target partition in an `ALTER PARTITION` clause.

DROP [DEFAULT] PARTITION

Drops the specified partition. If the partition has subpartitions, the subpartitions are automatically dropped as well.

TRUNCATE [DEFAULT] PARTITION

Truncates the specified partition. If the partition has subpartitions, the subpartitions are automatically truncated as well.

RENAME [DEFAULT] PARTITION

Changes the partition name of a partition (not the relation name). Partitioned tables are created using the naming convention: <*parentname*>_<*level*>_prt_<*partition_name*>.

ADD DEFAULT PARTITION

Adds a default partition to an existing partition design. When data does not match to an existing partition, it is inserted into the default partition. Partition designs that do not have a default partition will reject incoming rows that do not match to an existing partition. Default partitions must be given a name.

ADD PARTITION

partition_element - Using the existing partition type of the table (range or list), defines the boundaries of new partition you are adding.

name - A name for this new partition.

VALUES - For list partitions, defines the value(s) that the partition will contain.

START - For range partitions, defines the starting range value for the partition. By default, start values are `INCLUSIVE`. For example, if you declared a start date of '`2016-01-01`', then the partition would contain all dates greater than or equal to '`2016-01-01`'. Typically the data type of the `START` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

END - For range partitions, defines the ending range value for the partition. By default, end values are `EXCLUSIVE`. For example, if you declared an end date of '`2016-02-01`', then the partition would contain all dates less than but not equal to '`2016-02-01`'. Typically the data type of the `END` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

WITH - Sets the table storage options for a partition. For example, you may want older partitions to be append-optimized tables and newer partitions to be regular heap tables. See *CREATE TABLE* for a description of the storage options.

TABLESPACE - The name of the tablespace in which the partition is to be created.

subpartition_spec - Only allowed on partition designs that were created without a subpartition template. Declares a subpartition specification for the new partition you are adding. If the partitioned table was originally defined using a subpartition template, then the template will be used to generate the subpartitions automatically.

EXCHANGE [DEFAULT] PARTITION

Exchanges another table into the partition hierarchy into the place of an existing partition. In a multi-level partition design, you can only exchange the lowest level partitions (those that contain data).

The Greenplum Database server configuration parameter `gp_enable_exchange_default_partition` controls availability of the `EXCHANGE DEFAULT PARTITION` clause. The default value for the parameter is `off`. The clause is not available and Greenplum Database returns an error if the clause is specified in an `ALTER TABLE` command.

For information about the parameter, see *Server Configuration Parameters*.

> **Warning:** Before you exchange the default partition, you must ensure the data in the table to be exchanged, the new default partition, is valid for the

default partition. For example, the data in the new default partition must not contain data that would be valid in other leaf child partitions of the partitioned table. Otherwise, queries against the partitioned table with the exchanged default partition that are executed by GPORCA might return incorrect results.

WITH TABLE *table_name* - The name of the table you are swapping into the partition design. You can exchange a table where the table data is stored in the database. For example, the table is created with the `CREATE TABLE` command. The table must have the same number of columns, column order, column names, column types, and distribution policy as the parent table.

With the `EXCHANGE PARTITION` clause, you can also exchange a readable external table (created with the `CREATE EXTERNAL TABLE` command) into the partition hierarchy in the place of an existing leaf child partition. If you specify a readable external table, you must also specify the `WITHOUT VALIDATION` clause to skip table validation against the `CHECK` constraint of the partition you are exchanging.

Exchanging a leaf child partition with an external table is not supported if the partitioned table contains a column with a check constraint or a `NOT NULL` constraint.

You cannot exchange a partition with a replicated table. Exchanging a partition with a partitioned table or a child partition of a partitioned table is not supported.

WITH | **WITHOUT VALIDATION** - Validates that the data in the table matches the `CHECK` constraint of the partition you are exchanging. The default is to validate the data against the `CHECK` constraint.

Warning: If you specify the `WITHOUT VALIDATION` clause, you must ensure that the data in table that you are exchanging for an existing child leaf partition is valid against the `CHECK` constraints on the partition. Otherwise, queries against the partitioned table might return incorrect results.

SET SUBPARTITION TEMPLATE

Modifies the subpartition template for an existing partition. After a new subpartition template is set, all new partitions added will have the new subpartition design (existing partitions are not modified).

SPLIT DEFAULT PARTITION

Splits a default partition. In a multi-level partition, only a range partition can be split, not a list partition, and you can only split the lowest level default partitions (those that contain data). Splitting a default partition creates a new partition containing the values specified and leaves the default partition containing any values that do not match to an existing partition.

AT - For list partitioned tables, specifies a single list value that should be used as the criteria for the split.

START - For range partitioned tables, specifies a starting value for the new partition.

END - For range partitioned tables, specifies an ending value for the new partition.

INTO - Allows you to specify a name for the new partition. When using the `INTO` clause to split a default partition, the second partition name specified should always be that of the existing default partition. If you do not know the name of the default partition, you can look it up using the *pg_partitions* view.

SPLIT PARTITION

Splits an existing partition into two partitions. In a multi-level partition, only a range partition can be split, not a list partition, and you can only split the lowest level partitions (those that contain data).

AT - Specifies a single value that should be used as the criteria for the split. The partition will be divided into two new partitions with the split value specified being the starting range for the *latter* partition.

INTO - Allows you to specify names for the two new partitions created by the split.

partition_name

The given name of a partition. The given partition name is the `partitionname` column value in the *pg_partitions* system view.

FOR (RANK(number))

For range partitions, the rank of the partition in the range.

FOR ('*value*')

Specifies a partition by declaring a value that falls within the partition boundary specification. If the value declared with `FOR` matches to both a partition and one of its subpartitions (for example, if the value is a date and the table is partitioned by month and then by day), then `FOR` will operate on the first level where a match is found (for example, the monthly partition). If your intent is to operate on a subpartition, you must declare so as follows: `ALTER TABLE` *name* `ALTER PARTITION FOR ('2016-10-01') DROP PARTITION FOR ('2016-10-01');`

Notes

The table name specified in the `ALTER TABLE` command cannot be the name of a partition within a table.

Take special care when altering or dropping columns that are part of the Greenplum Database distribution key as this can change the distribution policy for the table.

Greenplum Database does not currently support foreign key constraints. For a unique constraint to be enforced in Greenplum Database, the table must be hash-distributed (not `DISTRIBUTED RANDOMLY`), and all of the distribution key columns must be the same as the initial columns of the unique constraint columns.

Adding a `CHECK` or `NOT NULL` constraint requires scanning the table to verify that existing rows meet the constraint, but does not require a table rewrite.

This table lists the `ALTER TABLE` operations that require a table rewrite when performed on tables defined with the specified type of table storage.

Table 82: ALTER TABLE Operations that Require Table Rewrite

Operation (See Note)	Append-Optimized, Column-Oriented	Append-Optimized	Heap
`ALTER COLUMN TYPE`	Yes	Yes	Yes
`ADD COLUMN`	No	Yes	Yes

Note: Dropping a system `oid` column also requires a table rewrite.

When a column is added with `ADD COLUMN`, all existing rows in the table are initialized with the column's default value, or `NULL` if no `DEFAULT` clause is specified. Adding a column with a non-null default or changing the type of an existing column will require the entire table and indexes to be rewritten. As an exception, if the `USING` clause does not change the column contents and the old type is either binary coercible to the new type or an unconstrained domain over the new type, a table rewrite is not needed, but any indexes on the affected columns must still be rebuilt. Table and/or index rebuilds may take a significant amount of time for a large table; and will temporarily require as much as double the disk space.

Important: The forms of `ALTER TABLE` that perform a table rewrite on an append-optimized table are not MVCC-safe. After a table rewrite, the table will appear empty to concurrent transactions if they are using a snapshot taken before the rewrite occurred. See *MVCC Caveats* for more details.

You can specify multiple changes in a single `ALTER TABLE` command, which will be done in a single pass over the table.

The `DROP COLUMN` form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated. If you drop the system `oid` column, however, the table is rewritten immediately.

To force immediate reclamation of space occupied by a dropped column, you can execute one of the forms of `ALTER TABLE` that performs a rewrite of the whole table. This results in reconstructing each row with the dropped column replaced by a null value.

The `USING` option of `SET DATA TYPE` can actually specify any expression involving the old values of the row; that is, it can refer to other columns as well as the one being converted. This allows very general conversions to be done with the `SET DATA TYPE` syntax. Because of this flexibility, the `USING` expression is not applied to the column's default value (if any); the result might not be a constant expression as required for a default. This means that when there is no implicit or assignment cast from old to new type, `SET DATA TYPE` might fail to convert the default even though a `USING` clause is supplied. In such cases, drop the default with `DROP DEFAULT`, perform the `ALTER TYPE`, and then use `SET DEFAULT` to add a suitable new default. Similar considerations apply to indexes and constraints involving the column.

If a table is partitioned or has any descendant tables, it is not permitted to add, rename, or change the type of a column, or rename an inherited constraint in the parent table without doing the same to the descendants. This ensures that the descendants always have columns matching the parent.

To see the structure of a partitioned table, you can use the view *`pg_partitions`*. This view can help identify the particular partitions you may want to alter.

A recursive `DROP COLUMN` operation will remove a descendant table's column only if the descendant does not inherit that column from any other parents and never had an independent definition of the column. A nonrecursive `DROP COLUMN` (`ALTER TABLE ONLY ... DROP COLUMN`) never removes any descendant columns, but instead marks them as independently defined rather than inherited.

The `TRIGGER`, `CLUSTER`, `OWNER`, and `TABLESPACE` actions never recurse to descendant tables; that is, they always act as though `ONLY` were specified. Adding a constraint recurses only for `CHECK` constraints that are not marked `NO INHERIT`.

These `ALTER PARTITION` operations are supported if no data is changed on a partitioned table that contains a leaf child partition that has been exchanged to use an external table. Otherwise, an error is returned.

- Adding or dropping a column.
- Changing the data type of column.

These `ALTER PARTITION` operations are not supported for a partitioned table that contains a leaf child partition that has been exchanged to use an external table:

- Setting a subpartition template.
- Altering the partition properties.
- Creating a default partition.
- Setting a distribution policy.
- Setting or dropping a `NOT NULL` constraint of column.
- Adding or dropping constraints.
- Splitting an external partition.

Changing any part of a system catalog table is not permitted.

Examples

Add a column to a table:

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

Rename an existing column:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

Rename an existing table:

```
ALTER TABLE distributors RENAME TO suppliers;
```

Add a not-null constraint to a column:

To rename an existing constraint:

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

Add a check constraint to a table and all of its children:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK
(char_length(zipcode) = 5);
```

To add a check constraint only to a table and not to its children:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) =
 5) NO INHERIT;
```

(The check constraint will not be inherited by future children, either.)

Remove a check constraint from a table and all of its children:

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

Remove a check constraint from one table only:

```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

(The check constraint remains in place for any child tables that inherit `distributors`.)

Move a table to a different schema:

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

Change the distribution policy of a table to replicated:

```
ALTER TABLE myschema.distributors SET DISTRIBUTED REPLICATED;
```

Add a new partition to a partitioned table:

```
ALTER TABLE sales ADD PARTITION
            START (date '2017-02-01') INCLUSIVE
            END (date '2017-03-01') EXCLUSIVE;
```

Add a default partition to an existing partition design:

```
ALTER TABLE sales ADD DEFAULT PARTITION other;
```

Rename a partition:

```
ALTER TABLE sales RENAME PARTITION FOR ('2016-01-01') TO
jan08;
```

Drop the first (oldest) partition in a range sequence:

```
ALTER TABLE sales DROP PARTITION FOR (RANK(1));
```

Exchange a table into your partition design:

```
ALTER TABLE sales EXCHANGE PARTITION FOR ('2016-01-01') WITH
TABLE jan08;
```

Split the default partition (where the existing default partition's name is `other`) to add a new monthly partition for January 2017:

```
ALTER TABLE sales SPLIT DEFAULT PARTITION
START ('2017-01-01') INCLUSIVE
END ('2017-02-01') EXCLUSIVE
INTO (PARTITION jan09, PARTITION other);
```

Split a monthly partition into two with the first partition containing dates January 1-15 and the second partition containing dates January 16-31:

```
ALTER TABLE sales SPLIT PARTITION FOR ('2016-01-01')
AT ('2016-01-16')
INTO (PARTITION jan081to15, PARTITION jan0816to31);
```

For a multi-level partitioned table that consists of three levels, year, quarter, and region, exchange a leaf partition `region` with the table `region_new`.

```
ALTER TABLE sales ALTER PARTITION year_1 ALTER PARTITION quarter_4 EXCHANGE
 PARTITION region WITH TABLE region_new ;
```

In the previous command, the two `ALTER PARTITION` clauses identify which `region` partition to exchange. Both clauses are required to identify the specific partition to exchange.

Compatibility

The forms `ADD` (without `USING INDEX`), `DROP`, `SET DEFAULT`, and `SET DATA TYPE` (without `USING`) conform with the SQL standard. The other forms are Greenplum Database extensions of the SQL standard. Also, the ability to specify more than one manipulation in a single `ALTER TABLE` command is an extension.

`ALTER TABLE DROP COLUMN` can be used to drop the only column of a table, leaving a zero-column table. This is an extension of SQL, which disallows zero-column tables.

See Also

`CREATE TABLE`, *`DROP TABLE`*

ALTER TABLESPACE

Changes the definition of a tablespace.

Synopsis

```
ALTER TABLESPACE name RENAME TO new_name

ALTER TABLESPACE name OWNER TO new_owner

ALTER TABLESPACE name SET ( tablespace_option = value [, ... ] )

ALTER TABLESPACE name RESET ( tablespace_option [, ... ] )
```

Description

`ALTER TABLESPACE` changes the definition of a tablespace.

You must own the tablespace to use `ALTER TABLESPACE`. To alter the owner, you must also be a direct or indirect member of the new owning role. (Note that superusers have these privileges automatically.)

Parameters

name

The name of an existing tablespace.

new_name

The new name of the tablespace. The new name cannot begin with *pg_* or *gp_* (reserved for system tablespaces).

new_owner

The new owner of the tablespace.

tablespace_parameter

A tablespace parameter to be set or reset. Currently, the only available parameters are *seq_page_cost* and *random_page_cost*. Setting either value for a particular tablespace will override the planner's usual estimate of the cost of reading pages from tables in that tablespace, as established by the configuration parameters of the same name (see *seq-page-cost*, *random-page-cost*). This may be useful if one tablespace is located on a disk which is faster or slower than the remainder of the I/O subsystem.

Examples

Rename tablespace `index_space` to `fast_raid`:

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

Change the owner of tablespace `index_space`:

```
ALTER TABLESPACE index_space OWNER TO mary;
```

Compatibility

There is no `ALTER TABLESPACE` statement in the SQL standard.

See Also

CREATE TABLESPACE, *DROP TABLESPACE*

ALTER TEXT SEARCH CONFIGURATION

Changes the definition of a text search configuration.

Synopsis

```
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary
 WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ]
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name
ALTER TEXT SEARCH CONFIGURATION name OWNER TO new_owner
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema
```

Description

`ALTER TEXT SEARCH CONFIGURATION` changes the definition of a text search configuration. You can modify its mappings from token types to dictionaries, or change the configuration's name or owner.

You must be the owner of the configuration to use `ALTER TEXT SEARCH CONFIGURATION`.

Parameters

name

The name (optionally schema-qualified) of an existing text search configuration.

token_type

The name of a token type that is emitted by the configuration's parser.

dictionary_name

The name of a text search dictionary to be consulted for the specified token type(s). If multiple dictionaries are listed, they are consulted in the specified order.

old_dictionary

The name of a text search dictionary to be replaced in the mapping.

new_dictionary

The name of a text search dictionary to be substituted for old_dictionary.

new_name

The new name of the text search configuration.

new_owner

The new owner of the text search configuration.

new_schema

The new schema for the text search configuration.

The `ADD MAPPING FOR` form installs a list of dictionaries to be consulted for the specified token type(s); it is an error if there is already a mapping for any of the token types. The `ALTER MAPPING FOR` form does the same, but first removing any existing mapping for those token types. The `ALTER MAPPING REPLACE` forms substitute *new_dictionary* for *old_dictionary* anywhere the latter appears. This is done for only the specified token types when `FOR` appears, or for all mappings of the configuration when it doesn't. The `DROP MAPPING` form removes all dictionaries for the specified token type(s), causing tokens of those types to be ignored by the text search configuration. It is an error if there is no mapping for the token types, unless `IF EXISTS` appears.

Examples

The following example replaces the `english` dictionary with the `swedish` dictionary anywhere that `english` is used within `my_config`.

```
ALTER TEXT SEARCH CONFIGURATION my_config
  ALTER MAPPING REPLACE english WITH swedish;
```

Compatibility

There is no `ALTER TEXT SEARCH CONFIGURATION` statement in the SQL standard.

See Also

`CREATE TEXT SEARCH CONFIGURATION`, *`DROP TEXT SEARCH CONFIGURATION`*

ALTER TEXT SEARCH DICTIONARY

Changes the definition of a text search dictionary.

Synopsis

```
ALTER TEXT SEARCH DICTIONARY name (
    option [ = value ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name
ALTER TEXT SEARCH DICTIONARY name OWNER TO new_owner
ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema
```

Description

`ALTER TEXT SEARCH DICTIONARY` changes the definition of a text search dictionary. You can change the dictionary's template-specific options, or change the dictionary's name or owner.

You must be the owner of the dictionary to use `ALTER TEXT SEARCH DICTIONARY`.

Parameters

name

The name (optionally schema-qualified) of an existing text search dictionary.

option

The name of a template-specific option to be set for this dictionary.

value

The new value to use for a template-specific option. If the equal sign and value are omitted, then any previous setting for the option is removed from the dictionary, allowing the default to be used.

new_name

The new name of the text search dictionary.

new_owner

The new owner of the text search dictionary.

new_schema

The new schema for the text search dictionary.

Template-specific options can appear in any order.

Examples

The following example command changes the stopword list for a Snowball-based dictionary. Other parameters remain unchanged.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian );
```

The following example command changes the language option to `dutch`, and removes the stopword option entirely.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch, StopWords );
```

The following example command "updates" the dictionary's definition without actually changing anything.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

(The reason this works is that the option removal code doesn't complain if there is no such option.) This trick is useful when changing configuration files for the dictionary: the ALTER will force existing database sessions to re-read the configuration files, which otherwise they would never do if they had read them earlier.

Compatibility

There is no `ALTER TEXT SEARCH DICTIONARY` statement in the SQL standard.

See Also

`CREATE TEXT SEARCH DICTIONARY`, *`DROP TEXT SEARCH DICTIONARY`*

ALTER TEXT SEARCH PARSER

Description

Changes the definition of a text search parser.

Synopsis

```
ALTER TEXT SEARCH PARSER name RENAME TO new_name
ALTER TEXT SEARCH PARSER name SET SCHEMA new_schema
```

Description

`ALTER TEXT SEARCH PARSER` changes the definition of a text search parser. Currently, the only supported functionality is to change the parser's name.

You must be a superuser to use `ALTER TEXT SEARCH PARSER`.

Parameters

name

The name (optionally schema-qualified) of an existing text search parser.

new_name

The new name of the text search parser.

new_schema

The new schema for the text search parser.

Compatibility

There is no `ALTER TEXT SEARCH PARSER` statement in the SQL standard.

See Also

CREATE TEXT SEARCH PARSER, *DROP TEXT SEARCH PARSER*

ALTER TEXT SEARCH TEMPLATE

Description

Changes the definition of a text search template.

Synopsis

```
ALTER TEXT SEARCH TEMPLATE name RENAME TO new_name
ALTER TEXT SEARCH TEMPLATE name SET SCHEMA new_schema
```

Description

`ALTER TEXT SEARCH TEMPLATE` changes the definition of a text search parser. Currently, the only supported functionality is to change the parser's name.

You must be a superuser to use `ALTER TEXT SEARCH TEMPLATE`.

Parameters

name

The name (optionally schema-qualified) of an existing text search template.

new_name

The new name of the text search template.

new_schema

The new schema for the text search template.

Compatibility

There is no `ALTER TEXT SEARCH TEMPLATE` statement in the SQL standard.

See Also

CREATE TEXT SEARCH TEMPLATE, *DROP TEXT SEARCH TEMPLATE*

ALTER TYPE

Changes the definition of a data type.

Synopsis

```
ALTER TYPE name action [, ... ]
ALTER TYPE name OWNER TO new_owner
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name
  [ CASCADE | RESTRICT ]
ALTER TYPE name RENAME TO new_name
ALTER TYPE name SET SCHEMA new_schema
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE |
  AFTER } existing_enum_value ]
ALTER TYPE name SET DEFAULT ENCODING ( storage_directive )

where action is one of:
```

```
 ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE |
RESTRICT ]
 DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
 ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type
[ COLLATE collation ] [ CASCADE | RESTRICT ]
```

where *storage_directive* is:

```
  COMPRESSTYPE={ZLIB | ZSTD | QUICKLZ | RLE_TYPE | NONE}
  COMPRESSLEVEL={0-19}
  BLOCKSIZE={8192-2097152}
```

Description

`ALTER TYPE` changes the definition of an existing type. There are several subforms:

- **`ADD ATTRIBUTE`** — Adds a new attribute to a composite type, using the same syntax as `CREATE TYPE`.
- **`DROP ATTRIBUTE [ IF EXISTS ]`** — Drops an attribute from a composite type. If `IF EXISTS` is specified and the attribute does not exist, no error is thrown. In this case a notice is issued instead.
- **`SET DATA TYPE`** — Changes the type of an attribute of a composite type.
- **`OWNER`** — Changes the owner of the type.
- **`RENAME`** — Changes the name of the type or the name of an individual attribute of a composite type.
- **`SET SCHEMA`** — Moves the type into another schema.
- **`ADD VALUE [ IF NOT EXISTS ] [ BEFORE | AFTER ]`** — Adds a new value to an enum type. The new value's place in the enum's ordering can be specified as being `BEFORE` or `AFTER` one of the existing values. Otherwise, the new item is added at the end of the list of values.

  If `IF NOT EXISTS` is specified, it is not an error if the type already contains the new value; a notice is issued but no other action is taken. Otherwise, an error will occur if the new value is already present.
- **`CASCADE`** — Automatically propagate the operation to typed tables of the type being altered, and their descendants.
- **`RESTRICT`** — Refuse the operation if the type being altered is the type of a typed table. This is the default.

The `ADD ATTRIBUTE`, `DROP ATTRIBUTE`, and `ALTER ATTRIBUTE` actions can be combined into a list of multiple alterations to apply in parallel. For example, it is possible to add several attributes and/or alter the type of several attributes in a single command.

You can change the name, the owner, and the schema of a type. You can also add or update storage options for a scalar type.

> **Note:** Greenplum Database does not support adding storage options for row or composite types.

You must own the type to use `ALTER TYPE`. To change the schema of a type, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the type's schema. (These restrictions enforce that altering the owner does not do anything that could be done by dropping and recreating the type. However, a superuser can alter ownership of any type.) To add an attribute or alter an attribute type, you must also have `USAGE` privilege on the data type.

`ALTER TYPE ... ADD VALUE` (the form that adds a new value to an enum type) cannot be executed inside a transaction block.

Comparisons involving an added enum value will sometimes be slower than comparisons involving only original members of the enum type. This will usually only occur if `BEFORE` or `AFTER` is used to set the new value's sort position somewhere other than at the end of the list. However, sometimes it will happen even though the new value is added at the end (this occurs if the OID counter "wrapped around" since the original creation of the enum type). The slowdown is usually insignificant; but if it matters, optimal

performance can be regained by dropping and recreating the enum type, or by dumping and reloading the database.

Parameters

name

The name (optionally schema-qualified) of an existing type to alter.

new_name

The new name for the type.

new_owner

The user name of the new owner of the type.

new_schema

The new schema for the type.

attribute_name

The name of the attribute to add, alter, or drop.

new_attribute_name

The new name of the attribute to be renamed.

data_type

The data type of the attribute to add, or the new type of the attribute to alter.

new_enum_value

The new value to be added to an enum type's list of values. Like all enum literals, it needs to be quoted.

existing_enum_value

The existing enum value that the new value should be added immediately before or after in the enum type's sort ordering. Like all enum literals, it needs to be quoted.

storage_directive

Identifies default storage options for the type when specified in a table column definition. Options include `COMPRESSTYPE`, `COMPRESSLEVEL`, and `BLOCKSIZE`.

COMPRESSTYPE — Set to `ZLIB` (the default), `ZSTD`, `RLE_TYPE`, or `QUICKLZ`[1] to specify the type of compression used.

> **Note:** [1]QuickLZ compression is available only in the commercial release of Pivotal Greenplum Database.

COMPRESSLEVEL — For Zstd compression, set to an integer value from 1 (fastest compression) to 19 (highest compression ratio). For zlib compression, the valid range is from 1 to 9. The QuickLZ compression level can only be set to 1. For `RLE_TYPE`, the compression level can be set to an integer value from 1 (fastest compression) to 4 (highest compression ratio). The default compression level is 1.

BLOCKSIZE — Set to the size, in bytes, for each block in the column. The `BLOCKSIZE` must be between 8192 and 2097152 bytes, and be a multiple of 8192. The default block size is 32768.

> **Note:** *storage_directives* defined at the table- or column-level override the default storage options defined for a type.

Examples

To rename the data type named `electronic_mail`:

```
ALTER TYPE electronic_mail RENAME TO email;
```

To change the owner of the user-defined type `email` to `joe`:

```
ALTER TYPE email OWNER TO joe;
```

To change the schema of the user-defined type `email` to `customers`:

```
ALTER TYPE email SET SCHEMA customers;
```

To set or alter the compression type and compression level of the user-defined type named `int33`:

```
ALTER TYPE int33 SET DEFAULT ENCODING (compresstype=zlib, compresslevel=7);
```

To add a new attribute to a type:

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

To add a new value to an enum type in a particular sort position:

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

Compatibility

The variants to add and drop attributes are part of the SQL standard; the other variants are Greenplum Database extensions.

See Also

CREATE TYPE, *DROP TYPE*

ALTER USER

Changes the definition of a database role (user).

Synopsis

```
ALTER USER name RENAME TO newname

ALTER USER name SET config_parameter {TO | =} {value | DEFAULT}

ALTER USER name RESET config_parameter

ALTER USER name RESOURCE QUEUE {queue_name | NONE}

ALTER USER name RESOURCE GROUP {group_name | NONE}

ALTER USER name [ [WITH] option [ ... ] ]
```

where *option* can be:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | CREATEEXTTABLE | NOCREATEEXTTABLE
      [ ( attribute='value'[, ...] ) ]
           where attributes and value are:
           type='readable'|'writable'
           protocol='gpfdist'|'http'
    | INHERIT | NOINHERIT
```

```
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | CONNECTION LIMIT connlimit
    | [ENCRYPTED | UNENCRYPTED] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | [ DENY deny_point ]
    | [ DENY BETWEEN deny_point AND deny_point]
    | [ DROP DENY FOR deny_point ]
```

Description

`ALTER USER` is an alias for `ALTER ROLE`. See *ALTER ROLE* for more information.

Compatibility

The `ALTER USER` statement is a Greenplum Database extension. The SQL standard leaves the definition of users to the implementation.

See Also

ALTER ROLE

ALTER USER MAPPING

Changes the definition of a user mapping for a foreign server.

Synopsis

```
ALTER USER MAPPING FOR { username | USER | CURRENT_USER | PUBLIC }
    SERVER servername
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

Description

`ALTER USER MAPPING` changes the definition of a user mapping for a foreign server.

The owner of a foreign server can alter user mappings for that server for any user. Also, a user granted `USAGE` privilege on the server can alter a user mapping for their own user name.

Parameters

username

User name of the mapping. `CURRENT_USER` and `USER` match the name of the current user. `PUBLIC` is used to match all present and future user names in the system.

servername

Server name of the user mapping.

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

Change options for the user mapping. The new options override any previously specified options. `ADD`, `SET`, and `DROP` specify the action to perform. If no operation is explicitly specified, the default operation is `ADD`. Option names must be unique. Greenplum Database validates names and values using the server's foreign-data wrapper.

Examples

Change the password for user mapping `bob`, server `foo`:

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (SET password 'public');
```

Compatibility

`ALTER USER MAPPING` conforms to ISO/IEC 9075-9 (SQL/MED). There is a subtle syntax issue: The standard omits the `FOR` key word. Since both `CREATE USER MAPPING` and `DROP USER MAPPING` use `FOR` in analogous positions, Greenplum Database diverges from the standard here in the interest of consistency and interoperability.

See Also

`CREATE USER MAPPING`, *`DROP USER MAPPING`*

ALTER VIEW

Changes properties of a view.

Synopsis

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET
 DEFAULT expression

ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT

ALTER VIEW [ IF EXISTS ] name OWNER TO new_owner

ALTER VIEW [ IF EXISTS ] name RENAME TO new_name

ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema

ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value]
 [, ... ] )

ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

Description

`ALTER VIEW` changes various auxiliary properties of a view. (If you want to modify the view's defining query, use `CREATE OR REPLACE VIEW`.

To execute this command you must be the owner of the view. To change a view's schema you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the view's schema. These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the view. However, a superuser can alter ownership of any view.

Parameters

name

The name (optionally schema-qualified) of an existing view.

`IF EXISTS`

Do not throw an error if the view does not exist. A notice is issued in this case.

`SET/DROP DEFAULT`

These forms set or remove the default value for a column. A view column's default value is substituted into any `INSERT` or `UPDATE` command whose target is the view, before applying any rules or triggers for the view. The view's default will therefore take precedence over any default values from underlying relations.

new_owner

The new owner for the view.

new_name

The new name of the view.

new_schema

The new schema for the view.

SET (*view_option_name* [= *view_option_value*] [, ...])
RESET (*view_option_name* [, ...])

Sets or resets a view option. Currently supported options are:

check_option (string)

Changes the check option of the view. The value must be `local` or `cascaded`.

security_barrier (boolean)

Changes the security-barrier property of the view. The value must be a Boolean value, such as `true` or `false`.

Notes

For historical reasons, `ALTER TABLE` can be used with views, too; however, the only variants of `ALTER TABLE` that are allowed with views are equivalent to the statements shown above.

Rename the view `myview` to `newview`:

```
ALTER VIEW myview RENAME TO newview;
```

Examples

To rename the view `foo` to `bar`:

```
ALTER VIEW foo RENAME TO bar;
```

To attach a default column value to an updatable view:

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1);  -- ts will receive a NULL
INSERT INTO a_view(id) VALUES(2);  -- ts will receive the current time
```

Compatibility

`ALTER VIEW` is a Greenplum Database extension of the SQL standard.

See Also

`CREATE VIEW`, *`DROP VIEW`* in the *Greenplum Database Utility Guide*

ANALYZE

Collects statistics about a database.

Synopsis

```
ANALYZE [VERBOSE] [table [ (column [, ...] ) ]]

ANALYZE [VERBOSE] {root_partition|leaf_partition} [ (column [, ...] )]

ANALYZE [VERBOSE] ROOTPARTITION {ALL | root_partition [ (column [, ...] )]}
```

Description

ANALYZE collects statistics about the contents of tables in the database, and stores the results in the system table *pg_statistic*. Subsequently, Greenplum Database uses these statistics to help determine the most efficient execution plans for queries. For information about the table statistics that are collected, see *Notes*.

With no parameter, ANALYZE collects statistics for every table in the current database. You can specify a table name to collect statistics for a single table. You can specify a set of column names, in which case the statistics only for those columns are collected.

ANALYZE does not collect statistics on external tables.

For partitioned tables, ANALYZE collects additional statistics, HyperLogLog (HLL) statistics, on the leaf child partitions. HLL statistics are used are used to derive number of distinct values (NDV) for queries against partitioned tables.

- When aggregating NDV estimates across multiple leaf child partitions, HLL statistics generate a more accurate NDV estimates than the standard table statistics.
- When updating HLL statistics, ANALYZE operations are required only on leaf child partitions that have changed. For example, ANALYZE is required if the leaf child partition data has changed, or if the leaf child partition has been exchanged with another table. For more information about updating partitioned table statistics, see *Notes*.

  Important: If you intend to execute queries on partitioned tables with GPORCA enabled (the default), then you must collect statistics on the root partition of the partitioned table with the ANALYZE or ANALYZE ROOTPARTITION command. For information about collecting statistics on partitioned tables and when the ROOTPARTITION keyword is required, see *Notes*. For information about GPORCA, see *Overview of GPORCA* in the *Greenplum Database Administrator Guide*.

  Note: You can also use the Greenplum Database utility `analyzedb` to update table statistics. The `analyzedb` utility can update statistics for multiple tables concurrently. The utility can also check table statistics and update statistics only if the statistics are not current or do not exist. For information about the utility, see the *Greenplum Database Utility Guide*.

Parameters

{ *root_partition* | *leaf_partition* } [(*column* [, ...])]

Collect statistics for partitioned tables including HLL statistics. HLL statistics are collected only on leaf child partitions.

ANALYZE *root_partition*, collects statistics on all leaf child partitions and the root partition.

ANALYZE *leaf_partition*, collects statistics on the leaf child partition.

By default, if you specify a leaf child partition, and all other leaf child partitions have statistics, ANALYZE updates the root partition statistics. If not all leaf child partitions have statistics, ANALYZE logs information about the leaf child partitions that do not have statistics. For information about when root partition statistics are collected, see *Notes*.

ROOTPARTITION [ALL]

Collect statistics only on the root partition of partitioned tables based on the data in the partitioned table. If possible, ANALYZE uses leaf child partition statistics to generate

root partition statistics. Otherwise, ANALYZE collects the statistics by sampling leaf child partition data. Statistics are not collected on the leaf child partitions, the data is only sampled. HLL statistics are not collected.

For information about when the ROOTPARTITION keyword is required, see *Notes*.

When you specify ROOTPARTITION, you must specify either ALL or the name of a partitioned table.

If you specify ALL with ROOTPARTITION, Greenplum Database collects statistics for the root partition of all partitioned tables in the database. If there are no partitioned tables in the database, a message stating that there are no partitioned tables is returned. For tables that are not partitioned tables, statistics are not collected.

If you specify a table name with ROOTPARTITION and the table is not a partitioned table, no statistics are collected for the table and a warning message is returned.

The ROOTPARTITION clause is not valid with VACUUM ANALYZE. The command VACUUM ANALYZE ROOTPARTITION returns an error.

The time to run ANALYZE ROOTPARTITION is similar to the time to analyze a non-partitioned table with the same data since ANALYZE ROOTPARTITION only samples the leaf child partition data.

For the partitioned table *sales_curr_yr*, this example command collects statistics only on the root partition of the partitioned table. ANALYZE ROOTPARTITION sales_curr_yr;

This example ANALYZE command collects statistics on the root partition of all the partitioned tables in the database.

```
ANALYZE ROOTPARTITION ALL;
```

VERBOSE

Enables display of progress messages. Enables display of progress messages. When specified, ANALYZE emits this information

- The table that is being processed.
- The query that is executed to generate the sample table.
- The column for which statistics is being computed.
- The queries that are issued to collect the different statistics for a single column.
- The statistics that are collected.

table

The name (possibly schema-qualified) of a specific table to analyze. If omitted, all regular tables (but not foreign tables) in the current database are analyzed.

column

The name of a specific column to analyze. Defaults to all columns.

Notes

Foreign tables are analyzed only when explicitly selected. Not all foreign data wrappers support ANALYZE. If the table's wrapper does not support ANALYZE, the command prints a warning and does nothing.

It is a good idea to run ANALYZE periodically, or just after making major changes in the contents of a table. Accurate statistics helps Greenplum Database choose the most appropriate query plan, and thereby improve the speed of query processing. A common strategy for read-mostly databases is to run *VACUUM* and ANALYZE once a day during a low-usage time of day. (This will not be sufficient if there is heavy update activity.) You can check for tables with missing statistics using the gp_stats_missing view, which is in the gp_toolkit schema:

```
SELECT * from gp_toolkit.gp_stats_missing;
```

ANALYZE requires SHARE UPDATE EXCLUSIVE lock on the target table. This lock conflicts with these locks: SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE.

If you run ANALYZE on a table that does not contain data, statistics are not collected for the table. For example, if you perform a TRUNCATE operation on a table that has statistics, and then run ANALYZE on the table, the statistics do not change.

For a partitioned table, specifying which portion of the table to analyze, the root partition or subpartitions (leaf child partition tables) can be useful if the partitioned table has a large number of partitions that have been analyzed and only a few leaf child partitions have changed.

> **Note:** When you create a partitioned table with the CREATE TABLE command, Greenplum Database creates the table that you specify (the root partition or parent table), and also creates a hierarchy of tables based on the partition hierarchy that you specified (the child tables).

- When you run ANALYZE on the root partitioned table, statistics are collected for all the leaf child partitions. Leaf child partitions are the lowest-level tables in the hierarchy of child tables created by Greenplum Database for use by the partitioned table.
- When you run ANALYZE on a leaf child partition, statistics are collected only for that leaf child partition and the root partition. If data in the leaf partition has changed (for example, you made significant updates to the leaf child partition data or you exchanged the leaf child partition), then you can run ANALYZE on the leaf child partition to collect table statistics. By default, if all other leaf child partitions have statistics, the command updates the root partition statistics.

  For example, if you collected statistics on a partitioned table with a large number partitions and then updated data in only a few leaf child partitions, you can run ANALYZE only on those partitions to update statistics on the partitions and the statistics on the root partition.
- When you run ANALYZE on a child table that is not a leaf child partition, statistics are not collected.

  For example, you can create a partitioned table with partitions for the years 2006 to 2016 and subpartitions for each month in each year. If you run ANALYZE on the child table for the year 2013 no statistics are collected. If you run ANALYZE on the leaf child partition for March of 2013, statistics are collected only for that leaf child partition.

For a partitioned table that contains a leaf child partition that has been exchanged to use an external table, ANALYZE does not collect statistics for the external table partition:

- If ANALYZE is run on an external table partition, the partition is not analyzed.
- If ANALYZE or ANALYZE ROOTPARTITION is run on the root partition, external table partitions are not sampled and root table statistics do not include external table partition.
- If the VERBOSE clause is specified, an informational message is displayed: `skipping external table`.

The Greenplum Database server configuration parameter *optimizer_analyze_root_partition* affects when statistics are collected on the root partition of a partitioned table. If the parameter is on (the default), the ROOTPARTITION keyword is not required to collect statistics on the root partition when you run ANALYZE. Root partition statistics are collected when you run ANALYZE on the root partition, or when you run ANALYZE on a child leaf partition of the partitioned table and the other child leaf partitions have statistics. If the parameter is `off`, you must run ANALZYE ROOTPARTITION to collect root partition statistics.

The statistics collected by ANALYZE usually include a list of some of the most common values in each column and a histogram showing the approximate data distribution in each column. One or both of these may be omitted if ANALYZE deems them uninteresting (for example, in a unique-key column, there are no common values) or if the column data type does not support the appropriate operators.

For large tables, ANALYZE takes a random sample of the table contents, rather than examining every row. This allows even very large tables to be analyzed in a small amount of time. Note, however, that the statistics are only approximate, and will change slightly each time ANALYZE is run, even if the actual table contents did not change. This may result in small changes in the planner's estimated costs

shown by EXPLAIN. In rare situations, this non-determinism will cause the query optimizer to choose a different query plan between runs of ANALYZE. To avoid this, raise the amount of statistics collected by ANALYZE by adjusting the *default_statistics_target* configuration parameter, or on a column-by-column basis by setting the per-column statistics target with ALTER TABLE ... ALTER COLUMN ... SET (n_distinct ...) (see ALTER TABLE). The target value sets the maximum number of entries in the most-common-value list and the maximum number of bins in the histogram. The default target value is 100, but this can be adjusted up or down to trade off accuracy of planner estimates against the time taken for ANALYZE and the amount of space occupied in pg_statistic. In particular, setting the statistics target to zero disables collection of statistics for that column. It may be useful to do that for columns that are never used as part of the WHERE, GROUP BY, or ORDER BY clauses of queries, since the planner will have no use for statistics on such columns.

The largest statistics target among the columns being analyzed determines the number of table rows sampled to prepare the statistics. Increasing the target causes a proportional increase in the time and space needed to do ANALYZE.

One of the values estimated by ANALYZE is the number of distinct values that appear in each column. Because only a subset of the rows are examined, this estimate can sometimes be quite inaccurate, even with the largest possible statistics target. If this inaccuracy leads to bad query plans, a more accurate value can be determined manually and then installed with ALTER TABLE ... ALTER COLUMN ... SET STATISTICS DISTINCT (see *ALTER TABLE*).

When Greenplum Database performs an ANALYZE operation to collect statistics for a table and detects that all the sampled table data pages are empty (do not contain valid data), Greenplum Database displays a message that a VACUUM FULL operation should be performed. If the sampled pages are empty, the table statistics will be inaccurate. Pages become empty after a large number of changes to the table, for example deleting a large number of rows. A VACUUM FULL operation removes the empty pages and allows an ANALYZE operation to collect accurate statistics.

If there are no statistics for the table, the server configuration parameter *gp_enable_relsize_collection* controls whether the Postgres Planner uses a default statistics file or estimates the size of a table using the pg_relation_size function. By default, the Postgres Planner uses the default statistics file to estimate the number of rows if statistics are not available.

Examples

Collect statistics for the table mytable:

```
ANALYZE mytable;
```

Compatibility

There is no ANALYZE statement in the SQL standard.

See Also

ALTER TABLE, *EXPLAIN*, *VACUUM*, analyzedb utility in the *Greenplum Database Utility Guide*.

BEGIN

Starts a transaction block.

Synopsis

```
BEGIN [WORK | TRANSACTION] [transaction_mode]
```

where *transaction_mode* is:

```
   ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ |
SERIALIZABLE}
   READ WRITE | READ ONLY
   [ NOT ] DEFERRABLE
```

Description

`BEGIN` initiates a transaction block, that is, all statements after a `BEGIN` command will be executed in a single transaction until an explicit `COMMIT` or `ROLLBACK` is given. By default (without `BEGIN`), Greenplum Database executes transactions in autocommit mode, that is, each statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done).

Statements are executed more quickly in a transaction block, because transaction start/commit requires significant CPU and disk activity. Execution of multiple statements inside a transaction is also useful to ensure consistency when making several related changes: other sessions will be unable to see the intermediate states wherein not all the related updates have been done.

If the isolation level, read/write mode, or deferrable mode is specified, the new transaction has those characteristics, as if *`SET TRANSACTION`* was executed.

Parameters

`WORK`
`TRANSACTION`

Optional key words. They have no effect.

`SERIALIZABLE`
`READ COMMITTED`
`READ UNCOMMITTED`

The SQL standard defines four transaction isolation levels: `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`.

`READ UNCOMMITTED` allows transactions to see changes made by uncomitted concurrent transactions. This is not possible in Greenplum Database, so `READ UNCOMMITTED` is treated the same as `READ COMMITTED`.

`READ COMMITTED`, the default isolation level in Greenplum Database, guarantees that a statement can only see rows committed before it began. The same statement executed twice in a transaction can produce different results if another concurrent transaction commits after the statement is executed the first time.

The `REPEATABLE READ` isolation level guarantees that a transaction can only see rows committed before it began. `REPEATABLE READ` is the strictest transaction isolation level Greenplum Database supports. Applications that use the `REPEATABLE READ` isolation level must be prepared to retry transactions due to serialization failures.

The `SERIALIZABLE` transaction isolation level guarantees that executing multiple concurrent transactions produces the same effects as running the same transactions one at a time. If you specify `SERIALIZABLE`, Greenplum Database falls back to `REPEATABLE READ`.

Specifying `DEFERRABLE` has no effect in Greenplum Database, but the syntax is supported for compatibility with PostgreSQL. A transaction can only be deferred if it is `READ ONLY` and `SERIALIZABLE`, and Greenplum Database does not support `SERIALIAZABLE` transactions.

Notes

START TRANSACTION has the same functionality as BEGIN.

Use *COMMIT* or *ROLLBACK* to terminate a transaction block.

Issuing BEGIN when already inside a transaction block will provoke a warning message. The state of the transaction is not affected. To nest transactions within a transaction block, use savepoints (see SAVEPOINT).

Examples

To begin a transaction block:

```
BEGIN;
```

To begin a transaction block with the repeatable read isolation level:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Compatibility

BEGIN is a Greenplum Database language extension. It is equivalent to the SQL-standard command *START TRANSACTION*.

DEFERRABLE *transaction_mode* is a Greenplum Database language extension.

Incidentally, the BEGIN key word is used for a different purpose in embedded SQL. You are advised to be careful about the transaction semantics when porting database applications.

See Also

COMMIT, *ROLLBACK*, *START TRANSACTION*, *SAVEPOINT*

CHECKPOINT

Forces a transaction log checkpoint.

Synopsis

```
CHECKPOINT
```

Description

A checkpoint is a point in the transaction log sequence at which all data files have been updated to reflect the information in the log. All data files will be flushed to disk.

The CHECKPOINT command forces an immediate checkpoint when the command is issued, without waiting for a regular checkpoint scheduled by the system. CHECKPOINT is not intended for use during normal operation.

If executed during recovery, the CHECKPOINT command will force a restartpoint rather than writing a new checkpoint.

Only superusers may call CHECKPOINT.

Compatibility

The CHECKPOINT command is a Greenplum Database extension.

CLOSE

Closes a cursor.

Synopsis

```
CLOSE cursor_name
```

Description

`CLOSE` frees the resources associated with an open cursor. After the cursor is closed, no subsequent operations are allowed on it. A cursor should be closed when it is no longer needed.

Every non-holdable open cursor is implicitly closed when a transaction is terminated by `COMMIT` or `ROLLBACK`. A holdable cursor is implicitly closed if the transaction that created it aborts via `ROLLBACK`. If the creating transaction successfully commits, the holdable cursor remains open until an explicit `CLOSE` is executed, or the client disconnects.

Parameters

cursor_name

The name of an open cursor to close.

Notes

Greenplum Database does not have an explicit `OPEN` cursor statement. A cursor is considered open when it is declared. Use the `DECLARE` statement to declare (and open) a cursor.

You can see all available cursors by querying the `pg_cursors` system view.

If a cursor is closed after a savepoint which is later rolled back, the `CLOSE` is not rolled back; that is the cursor remains closed.

Examples

Close the cursor `portala`:

```
CLOSE portala;
```

Compatibility

`CLOSE` is fully conforming with the SQL standard.

See Also

DECLARE, *FETCH*, *MOVE*

CLUSTER

Physically reorders a heap storage table on disk according to an index. Not a recommended operation in Greenplum Database.

Synopsis

```
CLUSTER indexname ON tablename

CLUSTER [VERBOSE] tablename
```

```
CLUSTER [VERBOSE]
```

Description

`CLUSTER` orders a heap storage table based on an index. `CLUSTER` is not supported on append-optmized storage tables. Clustering an index means that the records are physically ordered on disk according to the index information. If the records you need are distributed randomly on disk, then the database has to seek across the disk to get the records requested. If those records are stored more closely together, then the fetching from disk is more sequential. A good example for a clustered index is on a date column where the data is ordered sequentially by date. A query against a specific date range will result in an ordered fetch from the disk, which leverages faster sequential access.

Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered. That is, no attempt is made to store new or updated rows according to their index order. If you wish, you can periodically recluster by issuing the command again. Setting the table's `FILLFACTOR` storage parameter to less than 100% can aid in preserving cluster ordering during updates, because updated rows are kept on the same page if enough space is available there.

When a table is clustered using this command, Greenplum Database remembers on which index it was clustered. The form `CLUSTER` *tablename* reclusters the table on the same index that it was clustered before. You can use the `CLUSTER` or `SET WITHOUT CLUSTER` forms of *ALTER TABLE* to set the index to use for future cluster operations, or to clear any previous setting. `CLUSTER` without any parameter reclusters all previously clustered tables in the current database that the calling user owns, or all tables if called by a superuser. This form of `CLUSTER` cannot be executed inside a transaction block.

When a table is being clustered, an `ACCESS EXCLUSIVE` lock is acquired on it. This prevents any other database operations (both reads and writes) from operating on the table until the `CLUSTER` is finished.

Parameters

indexname

The name of an index.

VERBOSE

Prints a progress report as each table is clustered.

tablename

The name (optionally schema-qualified) of a table.

Notes

In cases where you are accessing single rows randomly within a table, the actual order of the data in the table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using `CLUSTER`. If you are requesting a range of indexed values from a table, or a single indexed value that has multiple rows that match, `CLUSTER` will help because once the index identifies the table page for the first row that matches, all other rows that match are probably already on the same table page, and so you save disk accesses and speed up the query.

`CLUSTER` can re-sort the table using either an index scan on the specified index, or (if the index is a b-tree) a sequential scan followed by sorting. It will attempt to choose the method that will be faster, based on planner cost parameters and available statistical information.

When an index scan is used, a temporary copy of the table is created that contains the table data in the index order. Temporary copies of each index on the table are created as well. Therefore, you need free space on disk at least equal to the sum of the table size and the index sizes.

When a sequential scan and sort is used, a temporary sort file is also created, so that the peak temporary space requirement is as much as double the table size, plus the index sizes. This method is often faster

than the index scan method, but if the disk space requirement is intolerable, you can disable this choice by temporarily setting the *enable_sort* configuration parameter to `off`.

It is advisable to set *maintenance_work_mem* configuration parameter to a reasonably large value (but not more than the amount of RAM you can dedicate to the `CLUSTER` operation) before clustering.

Because the query optimizer records statistics about the ordering of tables, it is advisable to run `ANALYZE` on the newly clustered table. Otherwise, the planner may make poor choices of query plans.

Because `CLUSTER` remembers which indexes are clustered, you can cluster the tables you want clustered manually the first time, then set up a periodic maintenance script that executes `CLUSTER` without any parameters, so that the desired tables are periodically reclustered.

Note: `CLUSTER` is not supported with append-optimized tables.

Examples

Cluster the table `employees` on the basis of its index `emp_ind`:

```
CLUSTER emp_ind ON emp;
```

Cluster a large table by recreating it and loading it in the correct index order:

```
CREATE TABLE newtable AS SELECT * FROM table ORDER BY column;
DROP table;
ALTER TABLE newtable RENAME TO table;
CREATE INDEX column_ix ON table (column);
VACUUM ANALYZE table;
```

Compatibility

There is no `CLUSTER` statement in the SQL standard.

See Also

CREATE TABLE AS, *CREATE INDEX*

COMMENT

Defines or changes the comment of an object.

Synopsis

```
COMMENT ON
{ TABLE object_name |
  COLUMN relation_name.column_name |
  AGGREGATE agg_name (agg_signature) |
  CAST (source_type AS target_type) |
  COLLATION object_name
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  EXTENSION object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION func_name ([[argmode] [argname] argtype [, ...]]) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
```

```
  OPERATOR CLASS object_name USING index_method |
  [PROCEDURAL] LANGUAGE object_name |
  RESOURCE GROUP object_name |
  RESOURCE QUEUE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  TABLESPACE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name }
IS 'text'
```

where *agg_signature* is:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
 ] argtype [ , ... ]
```

Description

`COMMENT` stores a comment about a database object. Only one comment string is stored for each object. To remove a comment, write `NULL` in place of the text string. Comments are automatically dropped when the object is dropped.

For most kinds of object, only the object's owner can set the comment. Roles don't have owners, so the rule for `COMMENT ON ROLE` is that you must be superuser to comment on a superuser role, or have the `CREATEROLE` privilege to comment on non-superuser roles. Of course, a superuser can comment on anything.

Comments can be easily retrieved with the psql meta-commands `\dd`, `\d+`, and `\l+`. Other user interfaces to retrieve comments can be built atop the same built-in functions that psql uses, namely `obj_description`, `col_description`, and `shobj_description`.

Parameters

object_name
relation_name.column_name
agg_name
constraint_name
func_name
operator_name
rule_name
trigger_name

The name of the object to be commented. Names of tables, aggregates, collations, conversions, domains, foreign tables, functions, indexes, operators, operator classes, operator families, sequences, text search objects, types, views, and materialized views can be schema-qualified. When commenting on a column, *relation_name* must refer to a table, view, materialized view, composite type, or foreign table.

Note: Greenplum Database does not support triggers.

source_type

The name of the source data type of the cast.

target_type

The name of the target data type of the cast.

argmode

The mode of a function or aggregate argument: either `IN`, `OUT`, `INOUT`, or `VARIADIC`. If omitted, the default is `IN`. Note that `COMMENT` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the `IN`, `INOUT`, and `VARIADIC` arguments.

argname

The name of a function or aggregate argument. Note that `COMMENT ON FUNCTION` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type of a function or aggregate argument.

large_object_oid

The OID of the large object.

> **Note:** Greenplum Database does not support the PostgreSQL *large object facility* for streaming user data that is stored in large-object structures.

left_type
right_type

The data type(s) of the operator's arguments (optionally schema-qualified). Write `NONE` for the missing argument of a prefix or postfix operator.

PROCEDURAL

This is a noise word.

text

The new comment, written as a string literal; or `NULL` to drop the comment.

Notes

There is presently no security mechanism for viewing comments: any user connected to a database can see all the comments for objects in that database. For shared objects such as databases, roles, and tablespaces, comments are stored globally so any user connected to any database in the cluster can see all the comments for shared objects. Therefore, do not put security-critical information in comments.

Examples

Attach a comment to the table `mytable`:

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

Remove it again:

```
COMMENT ON TABLE mytable IS NULL;
```

Some more examples:

```
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample
 variance';
COMMENT ON CAST (text AS int4) IS 'Allow casts from text to int4';
COMMENT ON COLLATION "fr_CA" IS 'Canadian French';
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON CONVERSION my_conv IS 'Conversion to UTF8';
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Constrains column col';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON EXTENSION hstore IS 'implements the hstore data type';
```

```
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'my foreign data wrapper';
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Employee Information in other
 database';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON LANGUAGE plpython IS 'Python support for stored procedures';
COMMENT ON LARGE OBJECT 346344 IS 'Planning document';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR - (NONE, integer) IS 'Unary minus';
COMMENT ON OPERATOR CLASS int4ops USING btree IS '4 byte integer operators
 for btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'all integer operators
 for btrees';
COMMENT ON ROLE my_role IS 'Administration group for finance tables';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON SERVER myserver IS 'my foreign server';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TABLESPACE my_tablespace IS 'Tablespace for indexes';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Special word filtering';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Snowball stemmer for Swedish
 language';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Splits text into words';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Snowball stemmer';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

Compatibility

There is no `COMMENT` statement in the SQL standard.

COMMIT

Commits the current transaction.

Synopsis

```
COMMIT [WORK | TRANSACTION]
```

Description

`COMMIT` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Parameters

`WORK`

`TRANSACTION`

Optional key words. They have no effect.

Notes

Use *`ROLLBACK`* to abort a transaction.

Issuing `COMMIT` when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To commit the current transaction and make all changes permanent:

```
COMMIT;
```

Compatibility

The SQL standard only specifies the two forms `COMMIT` and `COMMIT WORK`. Otherwise, this command is fully conforming.

See Also

BEGIN, *END*, *START TRANSACTION*, *ROLLBACK*

COPY

Copies data between a file and a table.

Synopsis

```
COPY table_name [(column_name [, ...])]
     FROM {'filename' | PROGRAM 'command' | STDIN}
     [ [ WITH ] ( option [, ...] ) ]
     [ ON SEGMENT ]

COPY { table_name [(column_name [, ...])] | (query)}
     TO {'filename' | PROGRAM 'command' | STDOUT}
     [ [ WITH ] ( option [, ...] ) ]
     [ ON SEGMENT ]
```

where *option* can be one of:

```
FORMAT format_name
OIDS [ boolean ]
FREEZE [ boolean ]
DELIMITER 'delimiter_character'
NULL 'null string'
HEADER [ boolean ]
QUOTE 'quote_character'
ESCAPE 'escape_character'
FORCE_QUOTE { ( column_name [, ...] ) | * }
FORCE_NOT_NULL ( column_name [, ...] )
ENCODING 'encoding_name'
FILL MISSING FIELDS
LOG ERRORS [ SEGMENT REJECT LIMIT count [ ROWS | PERCENT ] ]
IGNORE EXTERNAL PARTITIONS
```

Description

`COPY` moves data between Greenplum Database tables and standard file-system files. `COPY TO` copies the contents of a table to a file (or multiple files based on the segment ID if copying `ON SEGMENT`), while `COPY FROM` copies data from a file to a table (appending the data to whatever is in the table already). `COPY TO` can also copy the results of a `SELECT` query.

If a list of columns is specified, `COPY` will only copy the data in the specified columns to or from the file. If there are any columns in the table that are not in the column list, `COPY FROM` will insert the default values for those columns.

COPY with a file name instructs the Greenplum Database master host to directly read from or write to a file. The file must be accessible to the master host and the name must be specified from the viewpoint of the master host.

When COPY is used with the ON SEGMENT clause, the COPY TO causes segments to create individual segment-oriented files, which remain on the segment hosts. The *filename* argument for ON SEGMENT takes the string literal <SEGID> (required) and uses either the absolute path or the <SEG_DATA_DIR> string literal. When the COPY operation is run, the segment IDs and the paths of the segment data directories are substituted for the string literal values.

Using COPY TO with a replicated table (DISTRIBUTED REPLICATED) as source creates a file with rows from a single segment so that the target file contains no duplicate rows. Using COPY TO with the ON SEGMENT clause with a replicated table as source creates target files on segment hosts containing all table rows.

The ON SEGMENT clause allows you to copy table data to files on segment hosts for use in operations such as migrating data between clusters or performing a backup. Segment data created by the ON SEGMENT clause can be restored by tools such as gpfdist, which is useful for high speed data loading.

Warning: Use of the ON SEGMENT clause is recommended for expert users only.

When PROGRAM is specified, the server executes the given command and reads from the standard output of the program, or writes to the standard input of the program. The command must be specified from the viewpoint of the server, and be executable by the gpadmin user.

When STDIN or STDOUT is specified, data is transmitted via the connection between the client and the master. STDIN and STDOUT cannot be used with the ON SEGMENT clause.

If SEGMENT REJECT LIMIT is used, then a COPY FROM operation will operate in single row error isolation mode. In this release, single row error isolation mode only applies to rows in the input file with format errors — for example, extra or missing attributes, attributes of a wrong data type, or invalid client encoding sequences. Constraint errors such as violation of a NOT NULL, CHECK, or UNIQUE constraint will still be handled in 'all-or-nothing' input mode. The user can specify the number of error rows acceptable (on a per-segment basis), after which the entire COPY FROM operation will be aborted and no rows will be loaded. The count of error rows is per-segment, not per entire load operation. If the per-segment reject limit is not reached, then all rows not containing an error will be loaded and any error rows discarded. To keep error rows for further examination, specify the LOG ERRORS clause to capture error log information. The error information and the row is stored internally in Greenplum Database.

Outputs

On successful completion, a COPY command returns a command tag of the form, where *count* is the number of rows copied:

```
COPY count
```

If running a COPY FROM command in single row error isolation mode, the following notice message will be returned if any rows were not loaded due to format errors, where *count* is the number of rows rejected:

```
NOTICE: Rejected count badly formatted rows.
```

Parameters

table_name

The name (optionally schema-qualified) of an existing table.

column_name

An optional list of columns to be copied. If no column list is specified, all columns of the table will be copied.

When copying in text format, the default, a row of data in a column of type bytea can be up to 256MB.

query

A SELECT or VALUES command whose results are to be copied. Note that parentheses are required around the query.

filename

The path name of the input or output file. An input file name can be an absolute or relative path, but an output file name must be an absolute path. Windows users might need to use an E'' string and double any backslashes used in the path name.

PROGRAM '*command*'

Specify a command to execute. In COPY FROM, the input is read from standard output of the command, and in COPY TO, the output is written to the standard input of the command. The *command* must be specified from the viewpoint of the Greenplum Database master host system, and must be executable by the Greenplum Database administrator user (gpadmin).

The *command* is invoked by a shell. When passing arguments to the shell, strip or escape any special characters that have a special meaning for the shell. For security reasons, it is best to use a fixed command string, or at least avoid passing any user input in the string.

When ON SEGMENT is specified, the command must be executable on all Greenplum Database primary segment hosts by the Greenplum Database administrator user (gpadmin). The command is executed by each Greenplum segment instance. The <SEGID> is required in the *command*.

See the ON SEGMENT clause for information about command syntax requirements and the data that is copied when the clause is specified.

STDIN

Specifies that input comes from the client application. The ON SEGMENT clause is not supported with STDIN.

STDOUT

Specifies that output goes to the client application. The ON SEGMENT clause is not supported with STDOUT.

boolean

Specifies whether the selected option should be turned on or off. You can write TRUE, ON, or 1 to enable the option, and FALSE, OFF, or 0 to disable it. The boolean value can also be omitted, in which case TRUE is assumed.

FORMAT

Selects the data format to be read or written: text, csv (Comma Separated Values), or binary. The default is text.

OIDS

Specifies copying the OID for each row. (An error is raised if OIDS is specified for a table that does not have OIDs, or in the case of copying a query.)

FREEZE

Requests copying the data with rows already frozen, just as they would be after running the VACUUM FREEZE command. This is intended as a performance option for initial data loading. Rows will be frozen only if the table being loaded has been created or truncated in the current subtransaction, there are no cursors open, and there are no older snapshots held by this transaction.

Note that all other sessions will immediately be able to see the data once it has been successfully loaded. This violates the normal rules of MVCC visibility and users specifying this option should be aware of the potential problems this might cause.

DELIMITER

Specifies the character that separates columns within each row (line) of the file. The default is a tab character in `text` format, a comma in `CSV` format. This must be a single one-byte character. This option is not allowed when using `binary` format.

`NULL`

Specifies the string that represents a null value. The default is `\N` (backslash-N) in `text` format, and an unquoted empty string in `CSV` format. You might prefer an empty string even in `text` format for cases where you don't want to distinguish nulls from empty strings. This option is not allowed when using `binary` format.

> **Note:** When using `COPY FROM`, any data item that matches this string will be stored as a null value, so you should make sure that you use the same string as you used with `COPY TO`.

`HEADER`

Specifies that a file contains a header line with the names of each column in the file. On output, the first line contains the column names from the table, and on input, the first line is ignored. This option is allowed only when using `CSV` format.

`QUOTE`

Specifies the quoting character to be used when a data value is quoted. The default is double-quote. This must be a single one-byte character. This option is allowed only when using `CSV` format.

`ESCAPE`

Specifies the character that should appear before a data character that matches the `QUOTE` value. The default is the same as the `QUOTE` value (so that the quoting character is doubled if it appears in the data). This must be a single one-byte character. This option is allowed only when using `CSV` format.

`FORCE_QUOTE`

Forces quoting to be used for all non-`NULL` values in each specified column. `NULL` output is never quoted. If `*` is specified, non-`NULL` values will be quoted in all columns. This option is allowed only in `COPY TO`, and only when using `CSV` format.

`FORCE_NOT_NULL`

Do not match the specified columns' values against the null string. In the default case where the null string is empty, this means that empty values will be read as zero-length strings rather than nulls, even when they are not quoted. This option is allowed only in `COPY FROM`, and only when using `CSV` format.

`ENCODING`

Specifies that the file is encoded in the *encoding_name*. If this option is omitted, the current client encoding is used. See the Notes below for more details.

`ON SEGMENT`

Specify individual, segment data files on the segment hosts. Each file contains the table data that is managed by the primary segment instance. For example, when copying data to files from a table with a `COPY TO...ON SEGMENT` command, the command creates a file on the segment host for each segment instance on the host. Each file contains the table data that is managed by the segment instance.

The `COPY` command does not copy data from or to mirror segment instances and segment data files.

The keywords `STDIN` and `STDOUT` are not supported with `ON SEGMENT`.

The <SEG_DATA_DIR> and <SEGID> string literals are used to specify an absolute path and file name with the following syntax:

```
COPY table [TO|FROM] '<SEG_DATA_DIR>/gpdumpname<SEGID>_suffix' ON
 SEGMENT;
```

<SEG_DATA_DIR>

The string literal representing the absolute path of the segment instance data directory for ON SEGMENT copying. The angle brackets (< and >) are part of the string literal used to specify the path. COPY replaces the string literal with the segment path(s) when COPY is run. An absolute path can be used in place of the <SEG_DATA_DIR> string literal.

<SEGID>

The string literal representing the content ID number of the segment instance to be copied when copying ON SEGMENT. <SEGID> is a required part of the file name when ON SEGMENT is specified. The angle brackets are part of the string literal used to specify the file name.

With COPY TO, the string literal is replaced by the content ID of the segment instance when the COPY command is run.

With COPY FROM, specify the segment instance content ID in the name of the file and place that file on the segment instance host. There must be a file for each primary segment instance on each host. When the COPY FROM command is run, the data is copied from the file to the segment instance.

When the PROGRAM *command* clause is specified, the <SEGID> string literal is required in the *command*, the <SEG_DATA_DIR> string literal is optional. See *Examples*.

For a COPY FROM...ON SEGMENT command, the table distribution policy is checked when data is copied into the table. By default, an error is returned if a data row violates the table distribution policy. You can disable the distribution policy check with the server configuration parameter gp_enable_segment_copy_checking. See *Notes*.

NEWLINE

Specifies the newline used in your data files — LF (Line feed, 0x0A), CR (Carriage return, 0x0D), or CRLF (Carriage return plus line feed, 0x0D 0x0A). If not specified, a Greenplum Database segment will detect the newline type by looking at the first row of data it receives and using the first newline type encountered.

CSV

Selects Comma Separated Value (CSV) mode. See *CSV Format*.

FILL MISSING FIELDS

In COPY FROM more for both TEXT and CSV, specifying FILL MISSING FIELDS will set missing trailing field values to NULL (instead of reporting an error) when a row of data has missing data fields at the end of a line or row. Blank rows, fields with a NOT NULL constraint, and trailing delimiters on a line will still report an error.

LOG ERRORS

This is an optional clause that can precede a SEGMENT REJECT LIMIT clause to capture error log information about rows with formatting errors.

Error log information is stored internally and is accessed with the Greenplum Database built-in SQL function gp_read_error_log().

See *Notes* for information about the error log information and built-in functions for viewing and managing error log information.

SEGMENT REJECT LIMIT count [ROWS | PERCENT]

Runs a COPY FROM operation in single row error isolation mode. If the input rows have format errors they will be discarded provided that the reject limit count is not reached on

any Greenplum Database segment instance during the load operation. The reject limit count can be specified as number of rows (the default) or percentage of total rows (1-100). If PERCENT is used, each segment starts calculating the bad row percentage only after the number of rows specified by the parameter gp_reject_percent_threshold has been processed. The default for gp_reject_percent_threshold is 300 rows. Constraint errors such as violation of a NOT NULL, CHECK, or UNIQUE constraint will still be handled in 'all-or-nothing' input mode. If the limit is not reached, all good rows will be loaded and any error rows discarded.

Note: Greenplum Database limits the initial number of rows that can contain formatting errors if the SEGMENT REJECT LIMIT is not triggered first or is not specified. If the first 1000 rows are rejected, the COPY operation is stopped and rolled back.

The limit for the number of initial rejected rows can be changed with the Greenplum Database server configuration parameter gp_initial_bad_row_limit. See *Server Configuration Parameters* for information about the parameter.

IGNORE EXTERNAL PARTITIONS

When copying data from partitioned tables, data are not copied from leaf child partitions that are external tables. A message is added to the log file when data are not copied.

If this clause is not specified and Greenplum Database attempts to copy data from a leaf child partition that is an external table, an error is returned.

See the next section "Notes" for information about specifying an SQL query to copy data from leaf child partitions that are external tables.

Notes

COPY can only be used with tables, not with external tables or views. However, you can write COPY (SELECT * FROM viewname) TO ...

COPY only deals with the specific table named; it does not copy data to or from child tables. Thus for example COPY *table* TO shows the same data as SELECT * FROM ONLY table. But COPY (SELECT * FROM *table*) TO ... can be used to dump all of the data in an inheritance hierarchy.

Similarly, to copy data from a partitioned table with a leaf child partition that is an external table, use an SQL query to select the data to copy. For example, if the table my_sales contains a leaf child partition that is an external table, this command COPY my_sales TO stdout returns an error. This command sends the data to stdout:

```
COPY (SELECT * from my_sales ) TO stdout
```

The BINARY keyword causes all data to be stored/read as binary format rather than as text. It is somewhat faster than the normal text mode, but a binary-format file is less portable across machine architectures and Greenplum Database versions. Also, you cannot run COPY FROM in single row error isolation mode if the data is in binary format.

You must have SELECT privilege on the table whose values are read by COPY TO, and INSERT privilege on the table into which values are inserted by COPY FROM. It is sufficient to have column privileges on the columns listed in the command.

Files named in a COPY command are read or written directly by the database server, not by the client application. Therefore, they must reside on or be accessible to the Greenplum Database master host machine, not the client. They must be accessible to and readable or writable by the Greenplum Database system user (the user ID the server runs as), not the client. Only database superusers are permitted to name files with COPY, because this allows reading or writing any file that the server has privileges to access.

`COPY FROM` will invoke any triggers and check constraints on the destination table. However, it will not invoke rewrite rules. Note that in this release, violations of constraints are not evaluated for single row error isolation mode.

`COPY` input and output is affected by `DateStyle`. To ensure portability to other Greenplum Database installations that might use non-default `DateStyle` settings, `DateStyle` should be set to ISO before using `COPY TO`. It is also a good idea to avoid dumping data with *IntervalStyle* set to `sql_standard`, because negative interval values might be misinterpreted by a server that has a different setting for *IntervalStyle*.

Input data is interpreted according to `ENCODING` option or the current client encoding, and output data is encoded in `ENCODING` or the current client encoding, even if the data does not pass through the client but is read from or written to a file directly by the server.

When copying XML data from a file in text mode, the server configuration parameter *`xmloption`* affects the validation of the XML data that is copied. If the value is `content` (the default), XML data is validated as an XML content fragment. If the parameter value is `document`, XML data is validated as an XML document. If the XML data is not valid, `COPY` returns an error.

By default, `COPY` stops operation at the first error. This should not lead to problems in the event of a `COPY TO`, but the target table will already have received earlier rows in a `COPY FROM`. These rows will not be visible or accessible, but they still occupy disk space. This may amount to a considerable amount of wasted disk space if the failure happened well into a large `COPY FROM` operation. You may wish to invoke `VACUUM` to recover the wasted space. Another option would be to use single row error isolation mode to filter out error rows while still loading good rows.

When a `COPY FROM...ON SEGMENT` command is run, the server configuration parameter `gp_enable_segment_copy_checking` controls whether the table distribution policy (from the table `DISTRIBUTED` clause) is checked when data is copied into the table. The default is to check the distribution policy. An error is returned if the row of data violates the distribution policy for the segment instance. For information about the parameter, see *Server Configuration Parameters*.

Data from a table that is generated by a `COPY TO...ON SEGMENT` command can be used to restore table data with `COPY FROM...ON SEGMENT`. However, data restored to the segments is distributed according to the table distribution policy at the time the files were generated with the `COPY TO` command. The `COPY` command might return table distribution policy errors, if you attempt to restore table data and the table distribution policy was changed after the `COPY FROM...ON SEGMENT` was run.

> **Note:** If you run `COPY FROM...ON SEGMENT` and the server configuration parameter `gp_enable_segment_copy_checking` is `false`, manual redistribution of table data might be required. See the `ALTER TABLE` clause `WITH REORGANIZE`.

When you specify the `LOG ERRORS` clause, Greenplum Database captures errors that occur while reading the external table data. You can view and manage the captured error log data.

- Use the built-in SQL function `gp_read_error_log('`*`table_name`*`')`. It requires `SELECT` privilege on *table_name*. This example displays the error log information for data loaded into table `ext_expenses` with a `COPY` command:

  ```
  SELECT * from gp_read_error_log('ext_expenses');
  ```

  For information about the error log format, see *Viewing Bad Rows in the Error Log* in the *Greenplum Database Administrator Guide*.

  The function returns `FALSE` if *table_name* does not exist.
- If error log data exists for the specified table, the new error log data is appended to existing error log data. The error log information is not replicated to mirror segments.

- Use the built-in SQL function `gp_truncate_error_log('*table_name*')` to delete the error log data for *table_name*. It requires the table owner privilege This example deletes the error log information captured when moving data into the table `ext_expenses`:

  ```
  SELECT gp_truncate_error_log('ext_expenses');
  ```

  The function returns FALSE if *table_name* does not exist.

  Specify the `*` wildcard character to delete error log information for existing tables in the current database. Specify the string `*.*` to delete all database error log information, including error log information that was not deleted due to previous database issues. If * is specified, database owner privilege is required. If `*.*` is specified, operating system super-user privilege is required.

When a Greenplum Database user who is not a superuser runs a `COPY` command, the command can be controlled by a resource queue. The resource queue must be configured with the `ACTIVE_STATEMENTS` parameter that specifies a maximum limit on the number of queries that can be executed by roles assigned to that queue. Greenplum Database does not apply a cost value or memory value to a `COPY` command, resource queues with only cost or memory limits do not affect the running of `COPY` commands.

A non-superuser can run only these types of `COPY` commands:

- `COPY FROM` command where the source is `stdin`
- `COPY TO` command where the destination is `stdout`

For information about resource queues, see "Resource Management with Resource Queues" in the *Greenplum Database Administrator Guide*.

File Formats

File formats supported by `COPY`.

Text Format

When the `text` format is used, the data read or written is a text file with one line per table row. Columns in a row are separated by the *delimiter_character* (tab by default). The column values themselves are strings generated by the output function, or acceptable to the input function, of each attribute's data type. The specified null string is used in place of columns that are null. `COPY FROM` will raise an error if any line of the input file contains more or fewer columns than are expected. If `OIDS` is specified, the OID is read or written as the first column, preceding the user data columns.

The data file has two reserved characters that have special meaning to `COPY`:

- The designated delimiter character (tab by default), which is used to separate fields in the data file.
- A UNIX-style line feed (`\n` or `0x0a`), which is used to designate a new row in the data file. It is strongly recommended that applications generating `COPY` data convert data line feeds to UNIX-style line feeds rather than Microsoft Windows style carriage return line feeds (`\r\n` or `0x0a 0x0d`).

If your data contains either of these characters, you must escape the character so `COPY` treats it as data and not as a field separator or new row.

By default, the escape character is a \ (backslash) for text-formatted files and a " (double quote) for csv-formatted files. If you want to use a different escape character, you can do so using the `ESCAPE AS` clause. Make sure to choose an escape character that is not used anywhere in your data file as an actual data value. You can also disable escaping in text-formatted files by using `ESCAPE 'OFF'`.

For example, suppose you have a table with three columns and you want to load the following three fields using `COPY`.

- percentage sign = %
- vertical bar = |
- backslash = \

Your designated *delimiter_character* is | (pipe character), and your designated *escape* character is * (asterisk). The formatted row in your data file would look like this:

```
percentage sign = % | vertical bar = *| | backslash = \
```

Notice how the pipe character that is part of the data has been escaped using the asterisk character (*). Also notice that we do not need to escape the backslash since we are using an alternative escape character.

The following characters must be preceded by the escape character if they appear as part of a column value: the escape character itself, newline, carriage return, and the current delimiter character. You can specify a different escape character using the ESCAPE AS clause.

CSV Format

This format option is used for importing and exporting the Comma Separated Value (CSV) file format used by many other programs, such as spreadsheets. Instead of the escaping rules used by Greenplum Database standard text format, it produces and recognizes the common CSV escaping mechanism.

The values in each record are separated by the DELIMITER character. If the value contains the delimiter character, the QUOTE character, the ESCAPE character (which is double quote by default), the NULL string, a carriage return, or line feed character, then the whole value is prefixed and suffixed by the QUOTE character. You can also use FORCE_QUOTE to force quotes when outputting non-NULL values in specific columns.

The CSV format has no standard way to distinguish a NULL value from an empty string. Greenplum Database COPY handles this by quoting. A NULL is output as the NULL parameter string and is not quoted, while a non-NULL value matching the NULL string is quoted. For example, with the default settings, a NULL is written as an unquoted empty string, while an empty string data value is written with double quotes (""). Reading values follows similar rules. You can use FORCE_NOT_NULL to prevent NULL input comparisons for specific columns.

Because backslash is not a special character in the CSV format, \., the end-of-data marker, could also appear as a data value. To avoid any misinterpretation, a \. data value appearing as a lone entry on a line is automatically quoted on output, and on input, if quoted, is not interpreted as the end-of-data marker. If you are loading a file created by another application that has a single unquoted column and might have a value of \., you might need to quote that value in the input file.

> **Note:** In CSV format, all characters are significant. A quoted value surrounded by white space, or any characters other than DELIMITER, will include those characters. This can cause errors if you import data from a system that pads CSV lines with white space out to some fixed width. If such a situation arises you might need to preprocess the CSV file to remove the trailing white space, before importing the data into Greenplum Database.
>
> CSV format will both recognize and produce CSV files with quoted values containing embedded carriage returns and line feeds. Thus the files are not strictly one line per table row like text-format files
>
> **Note:** Many programs produce strange and occasionally perverse CSV files, so the file format is more a convention than a standard. Thus you might encounter some files that cannot be imported using this mechanism, and COPY might produce files that other programs cannot process.

Binary Format

The binary format option causes all data to be stored/read as binary format rather than as text. It is somewhat faster than the text and CSV formats, but a binary-format file is less portable across machine architectures and Greenplum Database versions. Also, the binary format is very data type specific; for example it will not work to output binary data from a smallint column and read it into an integer column, even though that would work fine in text format.

The binary file format consists of a file header, zero or more tuples containing the row data, and a file trailer. Headers and data are in network byte order.

- **File Header** — The file header consists of 15 bytes of fixed fields, followed by a variable-length header extension area. The fixed fields are:
  - **Signature** — 11-byte sequence PGCOPY\n\377\r\n\0 — note that the zero byte is a required part of the signature. (The signature is designed to allow easy identification of files that have been munged by a non-8-bit-clean transfer. This signature will be changed by end-of-line-translation filters, dropped zero bytes, dropped high bits, or parity changes.)
  - **Flags field** — 32-bit integer bit mask to denote important aspects of the file format. Bits are numbered from 0 (LSB) to 31 (MSB). Note that this field is stored in network byte order (most significant byte first), as are all the integer fields used in the file format. Bits 16-31 are reserved to denote critical file format issues; a reader should abort if it finds an unexpected bit set in this range. Bits 0-15 are reserved to signal backwards-compatible format issues; a reader should simply ignore any unexpected bits set in this range. Currently only one flag is defined, and the rest must be zero (Bit 16: 1 if data has OIDs, 0 if not).
  - **Header extension area length** — 32-bit integer, length in bytes of remainder of header, not including self. Currently, this is zero, and the first tuple follows immediately. Future changes to the format might allow additional data to be present in the header. A reader should silently skip over any header extension data it does not know what to do with. The header extension area is envisioned to contain a sequence of self-identifying chunks. The flags field is not intended to tell readers what is in the extension area. Specific design of header extension contents is left for a later release.
- **Tuples** — Each tuple begins with a 16-bit integer count of the number of fields in the tuple. (Presently, all tuples in a table will have the same count, but that might not always be true.) Then, repeated for each field in the tuple, there is a 32-bit length word followed by that many bytes of field data. (The length word does not include itself, and can be zero.) As a special case, -1 indicates a NULL field value. No value bytes follow in the NULL case.

  There is no alignment padding or any other extra data between fields.

  Presently, all data values in a binary-format file are assumed to be in binary format (format code one). It is anticipated that a future extension may add a header field that allows per-column format codes to be specified.

  If OIDs are included in the file, the OID field immediately follows the field-count word. It is a normal field except that it is not included in the field-count. In particular it has a length word — this will allow handling of 4-byte vs. 8-byte OIDs without too much pain, and will allow OIDs to be shown as null if that ever proves desirable.
- **File Trailer** — The file trailer consists of a 16-bit integer word containing `-1`. This is easily distinguished from a tuple's field-count word. A reader should report an error if a field-count word is neither `-1` nor the expected number of columns. This provides an extra check against somehow getting out of sync with the data.

Examples

Copy a table to the client using the vertical bar (|) as the field delimiter:

```
COPY country TO STDOUT (DELIMITER '|');
```

Copy data from a file into the `country` table:

```
COPY country FROM '/home/usr1/sql/country_data';
```

Copy into a file just the countries whose names start with 'A':

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO
'/home/usr1/sql/a_list_countries.copy';
```

Copy data from a file into the `sales` table using single row error isolation mode and log errors:

```
COPY sales FROM '/home/usr1/sql/sales_data' LOG ERRORS
   SEGMENT REJECT LIMIT 10 ROWS;
```

To copy segment data for later use, use the `ON SEGMENT` clause. Use of the `COPY TO ON SEGMENT` command takes the form:

```
COPY table TO '<SEG_DATA_DIR>/gpdumpname<SEGID>_suffix' ON SEGMENT;
```

The `<SEGID>` is required. However, you can substitute an absolute path for the `<SEG_DATA_DIR>` string literal in the path.

When you pass in the string literal `<SEG_DATA_DIR>` and `<SEGID>` to `COPY`, `COPY` will fill in the appropriate values when the operation is run.

For example, if you have `mytable` with the segments and mirror segments like this:

```
contentid | dbid | file segment location
    0     |  1   | /home/usr1/data1/gpsegdir0
    0     |  3   | /home/usr1/data_mirror1/gpsegdir0
    1     |  4   | /home/usr1/data2/gpsegdir1
    1     |  2   | /home/usr1/data_mirror2/gpsegdir1
```

running the command:

```
COPY mytable TO '<SEG_DATA_DIR>/gpbackup<SEGID>.txt' ON SEGMENT;
```

would result in the following files:

```
/home/usr1/data1/gpsegdir0/gpbackup0.txt
/home/usr1/data2/gpsegdir1/gpbackup1.txt
```

The content ID in the first column is the identifier inserted into the file path (for example, `gpsegdir0/gpbackup0.txt` above) Files are created on the segment hosts, rather than on the master, as they would be in a standard `COPY` operation. No data files are created for the mirror segments when using `ON SEGMENT` copying.

If an absolute path is specified, instead of `<SEG_DATA_DIR>`, such as in the statement

```
COPY mytable TO '/tmp/gpdir/gpbackup_<SEGID>.txt' ON SEGMENT;
```

files would be placed in `/tmp/gpdir` on every segment. The `gpfdist` tool can also be used to restore data files generated with `COPY TO` with the `ON SEGMENT` option if redistribution is necessary.

> **Note:** Tools such as `gpfdist` can be used to restore data. The backup/restore tools will not work with files that were manually generated with `COPY TO ON SEGMENT`.

This example uses a `SELECT` statement to copy data to files on each segment:

```
COPY (SELECT * FROM testtbl) TO '/tmp/mytst<SEGID>' ON SEGMENT;
```

This example copies the data from the `lineitem` table and uses the `PROGRAM` clause to add the data to the `/tmp/lineitem_program.csv` file with `cat` utility. The file is placed on the Greenplum Database master.

```
COPY LINEITEM TO PROGRAM 'cat > /tmp/lineitem.csv' CSV;
```

This example uses the PROGRAM and ON SEGMENT clauses to copy data to files on the segment hosts. On the segment hosts, the COPY command replaces <SEGID> with the segment content ID to create a file for each segment instance on the segment host.

```
COPY LINEITEM TO PROGRAM 'cat > /tmp/lineitem_program<SEGID>.csv' ON SEGMENT
 CSV;
```

This example uses the PROGRAM and ON SEGMENT clauses to copy data from files on the segment hosts. The COPY command replaces <SEGID> with the segment content ID when copying data from the files. On the segment hosts, there must be a file for each segment instance where the file name contains the segment content ID on the segment host.

```
COPY LINEITEM_4 FROM PROGRAM 'cat /tmp/lineitem_program<SEGID>.csv' ON
 SEGMENT CSV;
```

Compatibility

There is no COPY statement in the SQL standard.

The following syntax was used in earlier versions of Greenplum Database and is still supported:

```
COPY table_name [(column_name [, ...])] FROM {'filename' | PROGRAM 'command'
 | STDIN}
     [ [WITH]
       [ON SEGMENT]
       [BINARY]
       [OIDS]
       [HEADER]
       [DELIMITER [ AS ] 'delimiter_character']
       [NULL [ AS ] 'null string']
       [ESCAPE [ AS ] 'escape' | 'OFF']
       [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
       [CSV [QUOTE [ AS ] 'quote']
            [FORCE NOT NULL column_name [, ...]]
       [FILL MISSING FIELDS]
       [[LOG ERRORS]
       SEGMENT REJECT LIMIT count [ROWS | PERCENT] ]

COPY { table_name [(column_name [, ...])] | (query)} TO {'filename' |
 PROGRAM 'command' | STDOUT}
      [ [WITH]
        [ON SEGMENT]
        [BINARY]
        [OIDS]
        [HEADER]
        [DELIMITER [ AS ] 'delimiter_character']
        [NULL [ AS ] 'null string']
        [ESCAPE [ AS ] 'escape' | 'OFF']
        [CSV [QUOTE [ AS ] 'quote']
             [FORCE QUOTE column_name [, ...]] | * ]
      [IGNORE EXTERNAL PARTITIONS ]
```

Note that in this syntax, BINARY and CSV are treated as independent keywords, not as arguments of a FORMAT option.

See Also

CREATE EXTERNAL TABLE

CREATE AGGREGATE

Defines a new aggregate function.

Synopsis

```
CREATE AGGREGATE name ( [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = statefunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
  )

  CREATE AGGREGATE name ( [ [ argmode ] [ argname ] arg_data_type
 [ , ... ] ]
      ORDER BY [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = statefunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , HYPOTHETICAL ]
  )

  or the old syntax

  CREATE AGGREGATE name (
    BASETYPE = base_type,
    SFUNC = statefunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = minitial_condition ]
```

```
    [ , SORTOP = sort_operator ]
)
```

Description

`CREATE AGGREGATE` defines a new aggregate function. Some basic and commonly-used aggregate functions such as `count`, `min`, `max`, `sum`, `avg` and so on are already provided in Greenplum Database. If you define new types or need an aggregate function not already provided, you can use `CREATE AGGREGATE` to provide the desired features.

If a schema name is given (for example, `CREATE AGGREGATE myschema.myagg ...`) then the aggregate function is created in the specified schema. Otherwise it is created in the current schema.

An aggregate function is identified by its name and input data types. Two aggregate functions in the same schema can have the same name if they operate on different input types. The name and input data types of an aggregate function must also be distinct from the name and input data types of every ordinary function in the same schema. This behavior is identical to overloading of ordinary function names. See *CREATE FUNCTION*.

A simple aggregate function is made from one, two, or three ordinary functions (which must be `IMMUTABLE` functions):

- a state transition function *statefunc*
- an optional final calculation function *ffunc*
- an optional combine function *combinefunc*

These functions are used as follows:

```
statefunc( internal-state, next-data-values ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value
combinefunc( internal-state, internal-state ) ---> next-internal-state
```

Greenplum Database creates a temporary variable of data type *state_data_type* to hold the current internal state of the aggregate function. At each input row, the aggregate argument values are calculated and the state transition function is invoked with the current state value and the new argument values to calculate a new internal state value. After all the rows have been processed, the final function is invoked once to calculate the aggregate return value. If there is no final function then the ending state value is returned as-is.

> **Note:** If you write a user-defined aggregate in C, and you declare the state value (*state_data_type*) as type `internal`, there is a risk of an out-of-memory error occurring. If `internal` state values are not properly managed and a query acquires too much memory for state values, an out-of-memory error could occur. To prevent this, use `mpool_alloc(mpool, size)` to have Greenplum manage and allocate memory for non-temporary state values, that is, state values that have a lifespan for the entire aggregation. The argument *mpool* of the `mpool_alloc()` function is `aggstate->hhashtable->group_buf`. For an example, see the implementation of the numeric data type aggregates in `src/backend/utils/adt/numeric.c` in the Greenplum Database open source code.

You can specify *combinefunc* as a method for optimizing aggregate execution. By specifying *combinefunc*, the aggregate can be executed in parallel on segments first and then on the master. When a two-level execution is performed, the *statefunc* is executed on the segments to generate partial aggregate results, and *combinefunc* is executed on the master to aggregate the partial results from segments. If single-level aggregation is performed, all the rows are sent to the master and the *statefunc* is applied to the rows.

Single-level aggregation and two-level aggregation are equivalent execution strategies. Either type of aggregation can be implemented in a query plan. When you implement the functions `combinefunc` and `statefunc`, you must ensure that the invocation of the *statefunc* on the segment instances followed by

combinefunc on the master produce the same result as single-level aggregation that sends all the rows to the master and then applies only the *statefunc* to the rows.

An aggregate function can provide an optional initial condition, an initial value for the internal state value. This is specified and stored in the database as a value of type text, but it must be a valid external representation of a constant of the state value data type. If it is not supplied then the state value starts out NULL.

If *statefunc* is declared STRICT, then it cannot be called with NULL inputs. With such a transition function, aggregate execution behaves as follows. Rows with any null input values are ignored (the function is not called and the previous state value is retained). If the initial state value is NULL, then at the first row with all non-null input values, the first argument value replaces the state value, and the transition function is invoked at subsequent rows with all non-null input values. This is useful for implementing aggregates like max. Note that this behavior is only available when *state_data_type* is the same as the first *arg_data_type*. When these types are different, you must supply a non-null initial condition or use a nonstrict transition function.

If *statefunc* is not declared STRICT, then it will be called unconditionally at each input row, and must deal with NULL inputs and NULL state values for itself. This allows the aggregate author to have full control over the aggregate's handling of NULL values.

If the final function (*ffunc*) is declared STRICT, then it will not be called when the ending state value is NULL; instead a NULL result will be returned automatically. (This is the normal behavior of STRICT functions.) In any case the final function has the option of returning a NULL value. For example, the final function for avg returns NULL when it sees there were zero input rows.

Sometimes it is useful to declare the final function as taking not just the state value, but extra parameters corresponding to the aggregate's input values. The main reason for doing this is if the final function is polymorphic and the state value's data type would be inadequate to pin down the result type. These extra parameters are always passed as NULL (and so the final function must not be strict when the FINALFUNC_EXTRA option is used), but nonetheless they are valid parameters. The final function could for example make use of get_fn_expr_argtype to identify the actual argument type in the current call.

An aggregate can optionally support *moving-aggregate mode*, as described in *Moving-Aggregate Mode* in the PostgreSQL documentation. This requires specifying the *msfunc*, *minvfunc*, and *mstype* functions, and optionally the *mspace*, *mfinalfunc*, *mfinalfunc_extra*, and *minitcond* functions. Except for *minvfunc*, these functions work like the corresponding simple-aggregate functions without *m*; they define a separate implementation of the aggregate that includes an inverse transition function.

The syntax with ORDER BY in the parameter list creates a special type of aggregate called an *ordered-set aggregate*; or if HYPOTHETICAL is specified, then a *hypothetical-set aggregate* is created. These aggregates operate over groups of sorted values in order-dependent ways, so that specification of an input sort order is an essential part of a call. Also, they can have *direct* arguments, which are arguments that are evaluated only once per aggregation rather than once per input row. Hypothetical-set aggregates are a subclass of ordered-set aggregates in which some of the direct arguments are required to match, in number and data types, the aggregated argument columns. This allows the values of those direct arguments to be added to the collection of aggregate-input rows as an additional "hypothetical" row.

Single argument aggregate functions, such as min or max, can sometimes be optimized by looking into an index instead of scanning every input row. If this aggregate can be so optimized, indicate it by specifying a *sort operator*. The basic requirement is that the aggregate must yield the first element in the sort ordering induced by the operator; in other words:

```
SELECT agg(col) FROM tab;
```

must be equivalent to:

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

Further assumptions are that the aggregate function ignores NULL inputs, and that it delivers a NULL result if and only if there were no non-null inputs. Ordinarily, a data type's < operator is the proper sort operator for MIN, and > is the proper sort operator for MAX. Note that the optimization will never actually take effect unless the specified operator is the "less than" or "greater than" strategy member of a B-tree index operator class.

To be able to create an aggregate function, you must have USAGE privilege on the argument types, the state type(s), and the return type, as well as EXECUTE privilege on the transition and final functions.

Parameters

name

The name (optionally schema-qualified) of the aggregate function to create.

argmode

The mode of an argument: IN or VARIADIC. (Aggregate functions do not support OUT arguments.) If omitted, the default is IN. Only the last argument can be marked VARIADIC.

argname

The name of an argument. This is currently only useful for documentation purposes. If omitted, the argument has no name.

arg_data_type

An input data type on which this aggregate function operates. To create a zero-argument aggregate function, write * in place of the list of argument specifications. (An example of such an aggregate is count(*).)

base_type

In the old syntax for CREATE AGGREGATE, the input data type is specified by a basetype parameter rather than being written next to the aggregate name. Note that this syntax allows only one input parameter. To define a zero-argument aggregate function with this syntax, specify the basetype as "ANY" (not *). Ordered-set aggregates cannot be defined with the old syntax.

statefunc

The name of the state transition function to be called for each input row. For a normal N-argument aggregate function, the state transition function *statefunc* must take N+1 arguments, the first being of type *state_data_type* and the rest matching the declared input data types of the aggregate. The function must return a value of type *state_data_type*. This function takes the current state value and the current input data values, and returns the next state value.

For ordered-set (including hypothetical-set) aggregates, the state transition function *statefunc* receives only the current state value and the aggregated arguments, not the direct arguments. Otherwise it is the same.

state_data_type

The data type for the aggregate's state value.

state_data_size

The approximate average size (in bytes) of the aggregate's state value. If this parameter is omitted or is zero, a default estimate is used based on the *state_data_type*. The planner uses this value to estimate the memory required for a grouped aggregate query. Large values of this parameter discourage use of hash aggregation.

ffunc

The name of the final function called to compute the aggregate result after all input rows have been traversed. The function must take a single argument of type *state_data_type*. The return data type of the aggregate is defined as the return type of this function. If

ffunc is not specified, then the ending state value is used as the aggregate result, and the return type is *state_data_type*.

For ordered-set (including hypothetical-set) aggregates, the final function receives not only the final state value, but also the values of all the direct arguments.

If FINALFUNC_EXTRA is specified, then in addition to the final state value and any direct arguments, the final function receives extra NULL values corresponding to the aggregate's regular (aggregated) arguments. This is mainly useful to allow correct resolution of the aggregate result type when a polymorphic aggregate is being defined.

combinefunc

The name of a combine function. This is a function of two arguments, both of type *state_data_type*. It must return a value of *state_data_type*. A combine function takes two transition state values and returns a new transition state value representing the combined aggregation. In Greenplum Database, if the result of the aggregate function is computed in a segmented fashion, the combine function is invoked on the individual internal states in order to combine them into an ending internal state.

Note that this function is also called in hash aggregate mode within a segment. Therefore, if you call this aggregate function without a combine function, hash aggregate is never chosen. Since hash aggregate is efficient, consider defining a combine function whenever possible.

serialfunc

An aggregate function whose *state_data_type* is `internal` can participate in parallel aggregation only if it has a *serialfunc* function, which must serialize the aggregate state into a `bytea` value for transmission to another process. This function must take a single argument of type `internal` and return type `bytea`. A corresponding *deserialfunc* is also required.

deserialfunc

Deserialize a previously serialized aggregate state back into *state_data_type*. This function must take two arguments of types `bytea` and `internal`, and produce a result of type `internal`. (Note: the second, `internal` argument is unused, but is required for type safety reasons.)

initial_condition

The initial setting for the state value. This must be a string constant in the form accepted for the data type *state_data_type*. If not specified, the state value starts out null.

msfunc

The name of the forward state transition function to be called for each input row in moving-aggregate mode. This is exactly like the regular transition function, except that its first argument and result are of type *mstate_data_type*, which might be different from *state_data_type*.

minvfunc

The name of the inverse state transition function to be used in moving-aggregate mode. This function has the same argument and result types as *msfunc*, but it is used to remove a value from the current aggregate state, rather than add a value to it. The inverse transition function must have the same strictness attribute as the forward state transition function.

mstate_data_type

The data type for the aggregate's state value, when using moving-aggregate mode.

mstate_data_size

The approximate average size (in bytes) of the aggregate's state value, when using moving-aggregate mode. This works the same as *state_data_size*.

mffunc

The name of the final function called to compute the aggregate's result after all input rows have been traversed, when using moving-aggregate mode. This works the same as *ffunc*, except that its first argument's type is *mstate_data_type* and extra dummy arguments are specified by writing `MFINALFUNC_EXTRA`. The aggregate result type determined by *mffunc* or *mstate_data_type* must match that determined by the aggregate's regular implementation.

minitial_condition

The initial setting for the state value, when using moving-aggregate mode. This works the same as *initial_condition*.

sort_operator

The associated sort operator for a `MIN`- or `MAX`-like aggregate. This is just an operator name (possibly schema-qualified). The operator is assumed to have the same input data types as the aggregate (which must be a single-argument normal aggregate).

HYPOTHETICAL

For ordered-set aggregates only, this flag specifies that the aggregate arguments are to be processed according to the requirements for hypothetical-set aggregates: that is, the last few direct arguments must match the data types of the aggregated (`WITHIN GROUP`) arguments. The `HYPOTHETICAL` flag has no effect on run-time behavior, only on parse-time resolution of the data types and collations of the aggregate's arguments.

Notes

The ordinary functions used to define a new aggregate function must be defined first. Note that in this release of Greenplum Database, it is required that the *statefunc*, *ffunc*, and *combinefunc* functions used to create the aggregate are defined as `IMMUTABLE`.

If the value of the Greenplum Database server configuration parameter `gp_enable_multiphase_agg` is `off`, only single-level aggregation is performed.

Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files.

In previous versions of Greenplum Database, there was a concept of ordered aggregates. Since version 6, any aggregate can be called as an ordered aggregate, using the syntax:

```
name ( arg [ , ... ] [ORDER BY sortspec [ , ...]] )
```

The `ORDERED` keyword is accepted for backwards compatibility, but is ignored.

In previous versions of Greenplum Database, the `COMBINEFUNC` option was called `PREFUNC`. It is still accepted for backwards compatibility, as a synonym for `COMBINEFUNC`.

Example

The following simple example creates an aggregate function that computes the sum of two columns.

Before creating the aggregate function, create two functions that are used as the *`statefunc`* and *`combinefunc`* functions of the aggregate function.

This function is specified as the *`statefunc`* function in the aggregate function.

```
CREATE FUNCTION mysfunc_accum(numeric, numeric, numeric)
  RETURNS numeric
   AS 'select $1 + $2 + $3'
   LANGUAGE SQL
   IMMUTABLE
```

```
   RETURNS NULL ON NULL INPUT;
```

This function is specified as the `combinefunc` function in the aggregate function.

```
CREATE FUNCTION mycombine_accum(numeric, numeric )
  RETURNS numeric
   AS 'select $1 + $2'
   LANGUAGE SQL
   IMMUTABLE
   RETURNS NULL ON NULL INPUT;
```

This `CREATE AGGREGATE` command creates the aggregate function that adds two columns.

```
CREATE AGGREGATE agg_prefunc(numeric, numeric) (
   SFUNC = mysfunc_accum,
   STYPE = numeric,
   COMBINEFUNC = mycombine_accum,
   INITCOND = 0 );
```

The following commands create a table, adds some rows, and runs the aggregate function.

```
create table t1 (a int, b int) DISTRIBUTED BY (a);
insert into t1 values
   (10, 1),
   (20, 2),
   (30, 3);
select agg_prefunc(a, b) from t1;
```

This `EXPLAIN` command shows two phase aggregation.

```
explain select agg_prefunc(a, b) from t1;

QUERY PLAN
------------------------------------------------------------------------------
Aggregate (cost=1.10..1.11 rows=1 width=32)
 -> Gather Motion 2:1 (slice1; segments: 2) (cost=1.04..1.08 rows=1
      width=32)
     -> Aggregate (cost=1.04..1.05 rows=1 width=32)
       -> Seq Scan on t1 (cost=0.00..1.03 rows=2 width=8)
 (4 rows)
```

Compatibility

`CREATE AGGREGATE` is a Greenplum Database language extension. The SQL standard does not provide for user-defined aggregate functions.

See Also

`ALTER AGGREGATE`, *`DROP AGGREGATE`*, *`CREATE FUNCTION`*

CREATE CAST

Defines a new cast.

Synopsis

```
CREATE CAST (sourcetype AS targettype)
       WITH FUNCTION funcname (argtype [, ...])
       [AS ASSIGNMENT | AS IMPLICIT]
```

```
CREATE CAST (sourcetype AS targettype)
       WITHOUT FUNCTION
       [AS ASSIGNMENT | AS IMPLICIT]

CREATE CAST (sourcetype AS targettype)
       WITH INOUT
       [AS ASSIGNMENT | AS IMPLICIT]
```

Description

`CREATE CAST` defines a new cast. A cast specifies how to perform a conversion between two data types. For example,

```
SELECT CAST(42 AS float8);
```

converts the integer constant `42` to type `float8` by invoking a previously specified function, in this case `float8(int4)`. If no suitable cast has been defined, the conversion fails.

Two types may be binary coercible, which means that the types can be converted into one another without invoking any function. This requires that corresponding values use the same internal representation. For instance, the types `text` and `varchar` are binary coercible in both directions. Binary coercibility is not necessarily a symmetric relationship. For example, the cast from `xml` to `text` can be performed for free in the present implementation, but the reverse direction requires a function that performs at least a syntax check. (Two types that are binary coercible both ways are also referred to as binary compatible.)

You can define a cast as an *I/O conversion cast* by using the `WITH INOUT` syntax. An I/O conversion cast is performed by invoking the output function of the source data type, and passing the resulting string to the input function of the target data type. In many common cases, this feature avoids the need to write a separate cast function for conversion. An I/O conversion cast acts the same as a regular function-based cast; only the implementation is different.

By default, a cast can be invoked only by an explicit cast request, that is an explicit `CAST(x AS` *typename*`)` or `x::` *typename* construct.

If the cast is marked `AS ASSIGNMENT` then it can be invoked implicitly when assigning a value to a column of the target data type. For example, supposing that `foo.f1` is a column of type `text`, then:

```
INSERT INTO foo (f1) VALUES (42);
```

will be allowed if the cast from type `integer` to type `text` is marked `AS ASSIGNMENT`, otherwise not. The term *assignment cast* is typically used to describe this kind of cast.

If the cast is marked `AS IMPLICIT` then it can be invoked implicitly in any context, whether assignment or internally in an expression. The term *implicit cast* is typically used to describe this kind of cast. For example, consider this query:

```
SELECT 2 + 4.0;
```

The parser initially marks the constants as being of type `integer` and `numeric`, respectively. There is no `integer + numeric` operator in the system catalogs, but there is a `numeric + numeric` operator. This query succeeds if a cast from `integer` to `numeric` exists (it does) and is marked `AS IMPLICIT`, which in fact it is. The parser applies only the implicit cast and resolves the query as if it had been written as the following:

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

The catalogs also provide a cast from `numeric` to `integer`. If that cast were marked `AS IMPLICIT`, which it is not, then the parser would be faced with choosing between the above interpretation and the alternative of casting the `numeric` constant to `integer` and applying the `integer + integer`

operator. Lacking any knowledge of which choice to prefer, the parser would give up and declare the query ambiguous. The fact that only one of the two casts is implicit is the way in which we teach the parser to prefer resolution of a mixed `numeric`-and-`integer` expression as `numeric`; the parser has no built-in knowledge about that.

It is wise to be conservative about marking casts as implicit. An overabundance of implicit casting paths can cause Greenplum Database to choose surprising interpretations of commands, or to be unable to resolve commands at all because there are multiple possible interpretations. A good rule of thumb is to make a cast implicitly invokable only for information-preserving transformations between types in the same general type category. For example, the cast from `int2` to `int4` can reasonably be implicit, but the cast from `float8` to `int4` should probably be assignment-only. Cross-type-category casts, such as `text` to `int4`, are best made explicit-only.

> **Note:** Sometimes it is necessary for usability or standards-compliance reasons to provide multiple implicit casts among a set of types, resulting in ambiguity that cannot be avoided as described above. The parser uses a fallback heuristic based on type categories and preferred types that helps to provide desired behavior in such cases. See *CREATE TYPE* for more information.

To be able to create a cast, you must own the source or the target data type and have USAGE privilege on the other type. To create a binary-coercible cast, you must be superuser. (This restriction is made because an erroneous binary-coercible cast conversion can easily crash the server.)

Parameters

sourcetype

The name of the source data type of the cast.

targettype

The name of the target data type of the cast.

***funcname*(*argtype* [, ...])**

The function used to perform the cast. The function name may be schema-qualified. If it is not, Greenplum Database looks for the function in the schema search path. The function's result data type must match the target type of the cast.

Cast implementation functions may have one to three arguments. The first argument type must be identical to or binary-coercible from the cast's source type. The second argument, if present, must be type `integer`; it receives the type modifier associated with the destination type, or `-1` if there is none. The third argument, if present, must be type `boolean`; it receives `true` if the cast is an explicit cast, `false` otherwise. The SQL specification demands different behaviors for explicit and implicit casts in some cases. This argument is supplied for functions that must implement such casts. It is not recommended that you design your own data types this way.

The return type of a cast function must be identical to or binary-coercible to the cast's target type.

Ordinarily a cast must have different source and target data types. However, you are permitted to declare a cast with identical source and target types if it has a cast implementation function that takes more than one argument. This is used to represent type-specific length coercion functions in the system catalogs. The named function is used to coerce a value of the type to the type modifier value given by its second argument.

When a cast has different source and target types and a function that takes more than one argument, the cast converts from one type to another and applies a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two steps, one to convert between data types and a second to apply the modifier.

A cast to or from a domain type currently has no effect. Casting to or from a domain uses the casts associated with its underlying type.

WITHOUT FUNCTION

Indicates that the source type is binary-coercible to the target type, so no function is required to perform the cast.

WITH INOUT

Indicates that the cast is an I/O conversion cast, performed by invoking the output function of the source data type, and passing the resulting string to the input function of the target data type.

AS ASSIGNMENT

Indicates that the cast may be invoked implicitly in assignment contexts.

AS IMPLICIT

Indicates that the cast may be invoked implicitly in any context.

Notes

Note that in this release of Greenplum Database, user-defined functions used in a user-defined cast must be defined as `IMMUTABLE`. Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files.

Remember that if you want to be able to convert types both ways you need to declare casts both ways explicitly.

It is normally not necessary to create casts between user-defined types and the standard string types (`text`, `varchar`, and `char(`*n*`)`, as well as user-defined types that are defined to be in the string category). Greenplum Database provides automatic I/O conversion casts for these. The automatic casts to string types are treated as assignment casts, while the automatic casts from string types are explicit-only. You can override this behavior by declaring your own cast to replace an automatic cast, but usually the only reason to do so is if you want the conversion to be more easily invokable than the standard assignment-only or explicit-only setting. Another possible reason is that you want the conversion to behave differently from the type's I/O function - think twice before doing this. (A small number of the built-in types do indeed have different behaviors for conversions, mostly because of requirements of the SQL standard.)

It is recommended that you follow the convention of naming cast implementation functions after the target data type, as the built-in cast implementation functions are named. Many users are used to being able to cast data types using a function-style notation, that is *`typename`*`(x)`.

There are two cases in which a function-call construct is treated as a cast request without having matched it to an actual function. If a function call *`name(x)`* does not exactly match any existing function, but *`name`* is the name of a data type and `pg_cast` provides a binary-coercible cast to this type from the type of *`x`*, then the call will be construed as a binary-coercible cast. Greenplum Database makes this exception so that binary-coercible casts can be invoked using functional syntax, even though they lack any function. Likewise, if there is no `pg_cast` entry but the cast would be to or from a string type, the call is construed as an I/O conversion cast. This exception allows I/O conversion casts to be invoked using functional syntax.

There is an exception to the exception above: I/O conversion casts from composite types to string types cannot be invoked using functional syntax, but must be written in explicit cast syntax (either `CAST` or :: notation). This exception exists because after the introduction of automatically-provided I/O conversion casts, it was found to be too easy to accidentally invoke such a cast when you intended a function or column reference.

Examples

To create an assignment cast from type `bigint` to type `int4` using the function `int4(bigint)` (This cast is already predefined in the system.):

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

Compatibility

The `CREATE CAST` command conforms to the SQL standard, except that SQL does not make provisions for binary-coercible types or extra arguments to implementation functions. `AS IMPLICIT` is a Greenplum Database extension, too.

See Also

`CREATE FUNCTION`, *`CREATE TYPE`*, *`DROP CAST`*

CREATE COLLATION

Defines a new collation using the specified operating system locale settings, or by copying an existing collation.

Synopsis

```
CREATE COLLATION name (
    [ LOCALE = locale, ]
    [ LC_COLLATE = lc_collate, ]
    [ LC_CTYPE = lc_ctype ])

CREATE COLLATION name FROM existing_collation
```

Description

To be able to create a collation, you must have `CREATE` privilege on the destination schema.

Parameters

`name`

The name of the collation. The collation name can be schema-qualified. If it is not, the collation is defined in the current schema. The collation name must be unique within that schema. (The system catalogs can contain collations with the same name for other encodings, but these are ignored if the database encoding does not match.)

`locale`

This is a shortcut for setting `LC_COLLATE` and `LC_CTYPE` at once. If you specify this, you cannot specify either of those parameters.

`lc_collate`

Use the specified operating system locale for the `LC_COLLATE` locale category. The locale must be applicable to the current database encoding. (See *`CREATE DATABASE`* for the precise rules.)

`lc_ctype`

Use the specified operating system locale for the `LC_CTYPE` locale category. The locale must be applicable to the current database encoding. (See *`CREATE DATABASE`* for the precise rules.)

`existing_collation`

The name of an existing collation to copy. The new collation will have the same properties as the existing one, but it will be an independent object.

Notes

To be able to create a collation, you must have `CREATE` privilege on the destination schema.

Use `DROP COLLATION` to remove user-defined collations.

See *Collation Support* in the PostgreSQL documentation for more information about collation support in Greenplum Database.

Examples

To create a collation from the operating system locale `fr_FR.utf8` (assuming the current database encoding is `UTF8`):

```
CREATE COLLATION french (LOCALE = 'fr_FR.utf8');
```

To create a collation from an existing collation:

```
CREATE COLLATION german FROM "de_DE";
```

This can be convenient to be able to use operating-system-independent collation names in applications.

Compatibility

There is a `CREATE COLLATION` statement in the SQL standard, but it is limited to copying an existing collation. The syntax to create a new collation is a Greenplum Database extension.

See Also

ALTER COLLATION, *DROP COLLATION*

CREATE CONVERSION

Defines a new encoding conversion.

Synopsis

```
CREATE [DEFAULT] CONVERSION name FOR source_encoding TO
     dest_encoding FROM funcname
```

Description

`CREATE CONVERSION` defines a new conversion between character set encodings. Conversion names may be used in the convert function to specify a particular encoding conversion. Also, conversions that are marked `DEFAULT` can be used for automatic encoding conversion between client and server. For this purpose, two conversions, from encoding A to B and from encoding B to A, must be defined.

To create a conversion, you must have `EXECUTE` privilege on the function and `CREATE` privilege on the destination schema.

Parameters

`DEFAULT`

Indicates that this conversion is the default for this particular source to destination encoding. There should be only one default encoding in a schema for the encoding pair.

`name`

The name of the conversion. The conversion name may be schema-qualified. If it is not, the conversion is defined in the current schema. The conversion name must be unique within a schema.

`source_encoding`

The source encoding name.

`dest_encoding`

The destination encoding name.

funcname

The function used to perform the conversion. The function name may be schema-qualified. If it is not, the function will be looked up in the path. The function must have the following signature:

```
conv_proc(
    integer,  -- source encoding ID
    integer,  -- destination encoding ID
    cstring,  -- source string (null terminated C string)
    internal, -- destination (fill with a null terminated C
 string)
    integer   -- source string length
) RETURNS void;
```

Notes

Note that in this release of Greenplum Database, user-defined functions used in a user-defined conversion must be defined as `IMMUTABLE`. Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files.

Examples

To create a conversion from encoding `UTF8` to `LATIN1` using `myfunc`:

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

Compatibility

There is no `CREATE CONVERSION` statement in the SQL standard, but there is a `CREATE TRANSLATION` statement that is very similar in purpose and syntax.

See Also

ALTER CONVERSION, *CREATE FUNCTION*, *DROP CONVERSION*

CREATE DATABASE

Creates a new database.

Synopsis

```
CREATE DATABASE name [ [WITH] [OWNER [=] user_name]
                     [TEMPLATE [=] template]
                     [ENCODING [=] encoding]
                     [LC_COLLATE [=] lc_collate]
                     [LC_CTYPE [=] lc_ctype]
                     [TABLESPACE [=] tablespace]
                     [CONNECTION LIMIT [=] connlimit ] ]
```

Description

`CREATE DATABASE` creates a new database. To create a database, you must be a superuser or have the special `CREATEDB` privilege.

The creator becomes the owner of the new database by default. Superusers can create databases owned by other users by using the OWNER clause. They can even create databases owned by users with no special privileges. Non-superusers with CREATEDB privilege can only create databases owned by themselves.

By default, the new database will be created by cloning the standard system database template1. A different template can be specified by writing TEMPLATE *name*. In particular, by writing TEMPLATE template0, you can create a clean database containing only the standard objects predefined by Greenplum Database. This is useful if you wish to avoid copying any installation-local objects that may have been added to template1.

Parameters

name

The name of a database to create.

user_name

The name of the database user who will own the new database, or DEFAULT to use the default owner (the user executing the command).

template

The name of the template from which to create the new database, or DEFAULT to use the default template (*template1*).

encoding

Character set encoding to use in the new database. Specify a string constant (such as 'SQL_ASCII'), an integer encoding number, or DEFAULT to use the default encoding. For more information, see *Character Set Support*.

lc_collate

The collation order (LC_COLLATE) to use in the new database. This affects the sort order applied to strings, e.g. in queries with ORDER BY, as well as the order used in indexes on text columns. The default is to use the collation order of the template database. See the Notes section for additional restrictions.

lc_ctype

The character classification (LC_CTYPE) to use in the new database. This affects the categorization of characters, e.g. lower, upper and digit. The default is to use the character classification of the template database. See below for additional restrictions.

tablespace

The name of the tablespace that will be associated with the new database, or DEFAULT to use the template database's tablespace. This tablespace will be the default tablespace used for objects created in this database.

connlimit

The maximum number of concurrent connections possible. The default of -1 means there is no limitation.

Notes

CREATE DATABASE cannot be executed inside a transaction block.

When you copy a database by specifying its name as the template, no other sessions can be connected to the template database while it is being copied. New connections to the template database are locked out until CREATE DATABASE completes.

The CONNECTION LIMIT is not enforced against superusers.

The character set encoding specified for the new database must be compatible with the chosen locale settings (LC_COLLATE and LC_CTYPE). If the locale is C (or equivalently POSIX), then all encodings are

allowed, but for other locale settings there is only one encoding that will work properly. `CREATE DATABASE` will allow superusers to specify `SQL_ASCII` encoding regardless of the locale settings, but this choice is deprecated and may result in misbehavior of character-string functions if data that is not encoding-compatible with the locale is stored in the database.

The encoding and locale settings must match those of the template database, except when `template0` is used as template. This is because `COLLATE` and `CTYPE` affect the ordering in indexes, so that any indexes copied from the template database would be invalid in the new database with different settings. `template0`, however, is known to not contain any data or indexes that would be affected.

Examples

To create a new database:

```
CREATE DATABASE gpdb;
```

To create a database `sales` owned by user `salesapp` with a default tablespace of `salesspace`:

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

To create a database `music` which supports the ISO-8859-1 character set:

```
CREATE DATABASE music ENCODING 'LATIN1' TEMPLATE template0;
```

In this example, the `TEMPLATE template0` clause would only be required if `template1`'s encoding is not ISO-8859-1. Note that changing encoding might require selecting new `LC_COLLATE` and `LC_CTYPE` settings as well.

Compatibility

There is no `CREATE DATABASE` statement in the SQL standard. Databases are equivalent to catalogs, whose creation is implementation-defined.

See Also

ALTER DATABASE, *DROP DATABASE*

CREATE DOMAIN

Defines a new domain.

Synopsis

```
CREATE DOMAIN name [AS] data_type [DEFAULT expression]
       [ COLLATE collation ]
       [ CONSTRAINT constraint_name
       | NOT NULL | NULL
       | CHECK (expression) [...]]
```

Description

`CREATE DOMAIN` creates a new domain. A domain is essentially a data type with optional constraints (restrictions on the allowed set of values). The user who defines a domain becomes its owner. The domain name must be unique among the data types and domains existing in its schema.

If a schema name is given (for example, `CREATE DOMAIN myschema.mydomain ...`) then the domain is created in the specified schema. Otherwise it is created in the current schema.

Domains are useful for abstracting common constraints on fields into a single location for maintenance. For example, several tables might contain email address columns, all requiring the same CHECK constraint to verify the address syntax. It is easier to define a domain rather than setting up a column constraint for each table that has an email column.

To be able to create a domain, you must have USAGE privilege on the underlying type.

Parameters

name

The name (optionally schema-qualified) of a domain to be created.

data_type

The underlying data type of the domain. This may include array specifiers.

DEFAULT *expression*

Specifies a default value for columns of the domain data type. The value is any variable-free expression (but subqueries are not allowed). The data type of the default expression must match the data type of the domain. If no default value is specified, then the default value is the null value. The default expression will be used in any insert operation that does not specify a value for the column. If a default value is defined for a particular column, it overrides any default associated with the domain. In turn, the domain default overrides any default value associated with the underlying data type.

COLLATE *collation*

An optional collation for the domain. If no collation is specified, the underlying data type's default collation is used. The underlying type must be collatable if COLLATE is specified.

CONSTRAINT *constraint_name*

An optional name for a constraint. If not specified, the system generates a name.

NOT NULL

Values of this domain are normally prevented from being null. However, it is still possible for a domain with this constraint to take a null value if it is assigned a matching domain type that has become null, e.g. via a left outer join, or a command such as `INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false))`.

NULL

Values of this domain are allowed to be null. This is the default. This clause is only intended for compatibility with nonstandard SQL databases. Its use is discouraged in new applications.

CHECK *(expression)*

CHECK clauses specify integrity constraints or tests which values of the domain must satisfy. Each constraint must be an expression producing a Boolean result. It should use the key word VALUE to refer to the value being tested. Currently, CHECK expressions cannot contain subqueries nor refer to variables other than VALUE.

Examples

Create the us_zip_code data type. A regular expression test is used to verify that the value looks like a valid US zip code.

```
CREATE DOMAIN us_zip_code AS TEXT CHECK
       ( VALUE ~ '^\d{5}$' OR VALUE ~ '^\d{5}-\d{4}$' );
```

Compatibility

CREATE DOMAIN conforms to the SQL standard.

See Also

ALTER DOMAIN, *DROP DOMAIN*

CREATE EXTENSION

Registers an extension in a Greenplum database.

Synopsis

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
  [ WITH ] [ SCHEMA schema_name ]
           [ VERSION version ]
           [ FROM old_version ]
           [ CASCADE ]
```

Description

`CREATE EXTENSION` loads a new extension into the current database. There must not be an extension of the same name already loaded.

Loading an extension essentially amounts to running the extension script file. The script typically creates new SQL objects such as functions, data types, operators and index support methods. The `CREATE EXTENSION` command also records the identities of all the created objects, so that they can be dropped again if `DROP EXTENSION` is issued.

Loading an extension requires the same privileges that would be required to create the component extension objects. For most extensions this means superuser or database owner privileges are required. The user who runs `CREATE EXTENSION` becomes the owner of the extension for purposes of later privilege checks, as well as the owner of any objects created by the extension script.

Parameters

`IF NOT EXISTS`

Do not throw an error if an extension with the same name already exists. A notice is issued in this case. There is no guarantee that the existing extension is similar to the extension that would have been installed.

`extension_name`

The name of the extension to be installed. The name must be unique within the database. An extension is created from the details in the extension control file *SHAREDIR*`/extension/`*`extension_name`*`.control`.

SHAREDIR is the installation shared-data directory, for example `/usr/local/greenplum-db/share/postgresql`. The command `pg_config --sharedir` displays the directory.

`SCHEMA` ***`schema_name`***

The name of the schema in which to install the extension objects. This assumes that the extension allows its contents to be relocated. The named schema must already exist. If not specified, and the extension control file does not specify a schema, the current default object creation schema is used.

If the extension specifies a schema parameter in its control file, then that schema cannot be overridden with a `SCHEMA` clause. Normally, an error is raised if a `SCHEMA` clause is given and it conflicts with the extension schema parameter. However, if the `CASCADE` clause is also given, then *schema_name* is ignored when it conflicts. The given *schema_name* is used for the installation of any needed extensions that do not a specify schema in their control files.

The extension itself is not within any schema. Extensions have unqualified names that must be unique within the database. But objects belonging to the extension can be within a schema.

VERSION *version*

The version of the extension to install. This can be written as either an identifier or a string literal. The default version is value that is specified in the extension control file.

FROM *old_version*

Specify `FROM old_version` only if you are attempting to install an extension that replaces an *old-style* module that is a collection of objects that is not packaged into an extension. If specified, `CREATE EXTENSION` runs an alternative installation script that absorbs the existing objects into the extension, instead of creating new objects. Ensure that `SCHEMA` clause specifies the schema containing these pre-existing objects.

The value to use for *old_version* is determined by the extension author, and might vary if there is more than one version of the old-style module that can be upgraded into an extension. For the standard additional modules supplied with pre-9.1 PostgreSQL, specify `unpackaged` for the *old_version* when updating a module to extension style.

CASCADE

Automatically install dependent extensions are not already installed. Dependent extensions are checked recursively and those dependencies are also installed automatically. If the *SCHEMA* clause is specified, the schema applies to the extension and all dependent extensions that are installed. Other options that are specified are not applied to the automatically-installed dependent extensions. In particular, default versions are always selected when installing dependent extensions.

Notes

The extensions currently available for loading can be identified from the *pg_available_extensions* or *pg_available_extension_versions* system views.

Before you use `CREATE EXTENSION` to load an extension into a database, the supporting extension files must be installed including an extension control file and at least one least one SQL script file. The support files must be installed in the same location on all Greenplum Database hosts. For information about creating new extensions, see PostgreSQL information about *Packaging Related Objects into an Extension*.

Compatibility

`CREATE EXTENSION` is a Greenplum Database extension.

See Also

ALTER EXTENSION, *DROP EXTENSION*

CREATE EXTERNAL TABLE

Defines a new external table.

Synopsis

```
CREATE [READABLE] EXTERNAL [TEMPORARY | TEMP] TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION ('file://seghost[:port]/path/file' [, ...])
       | ('gpfdist://filehost[:port]/file_pattern[#transform=trans_name]'
           [, ...]
```

```
       | ('gpfdists://filehost[:port]/file_pattern[#transform=trans_name]'
           [, ...])
       | ('pxf://path-to-data?PROFILE=profile_name[&SERVER=server_name]
[&custom-option=value[...]]'))
       | ('s3://S3_endpoint[:port]/bucket_name/[S3_prefix] [region=S3-
region] [config=config_file]')
     [ON MASTER]
     FORMAT 'TEXT'
           [( [HEADER]
              [DELIMITER [AS] 'delimiter' | 'OFF']
              [NULL [AS] 'null string']
              [ESCAPE [AS] 'escape' | 'OFF']
              [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
              [FILL MISSING FIELDS] )]
         | 'CSV'
           [( [HEADER]
              [QUOTE [AS] 'quote']
              [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [FORCE NOT NULL column [, ...]]
              [ESCAPE [AS] 'escape']
              [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
              [FILL MISSING FIELDS] )]
         | 'CUSTOM' (Formatter=<formatter_specifications>)
    [ ENCODING 'encoding' ]
      [ [LOG ERRORS [PERSISTENTLY]] SEGMENT REJECT LIMIT count
      [ROWS | PERCENT] ]

CREATE [READABLE] EXTERNAL WEB [TEMPORARY | TEMP] TABLE table_name
   ( column_name data_type [, ...] | LIKE other_table )
      LOCATION ('http://webhost[:port]/path/file' [, ...])
    | EXECUTE 'command' [ON ALL
                         | MASTER
                         | number_of_segments
                         | HOST ['segment_hostname']
                         | SEGMENT segment_id ]
      FORMAT 'TEXT'
            [( [HEADER]
               [DELIMITER [AS] 'delimiter' | 'OFF']
               [NULL [AS] 'null string']
               [ESCAPE [AS] 'escape' | 'OFF']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
          | 'CSV'
            [( [HEADER]
               [QUOTE [AS] 'quote']
               [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [FORCE NOT NULL column [, ...]]
               [ESCAPE [AS] 'escape']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
          | 'CUSTOM' (Formatter=<formatter specifications>)
     [ ENCODING 'encoding' ]
     [ [LOG ERRORS [PERSISTENTLY]] SEGMENT REJECT LIMIT count
       [ROWS | PERCENT] ]

CREATE WRITABLE EXTERNAL [TEMPORARY | TEMP] TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION('gpfdist://outputhost[:port]/filename[#transform=trans_name]'
          [, ...])
      | ('gpfdists://outputhost[:port]/file_pattern[#transform=trans_name]'
          [, ...])
      FORMAT 'TEXT'
```

```
              [( [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [ESCAPE [AS] 'escape' | 'OFF'] )]
         | 'CSV'
              [([QUOTE [AS] 'quote']
              [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [FORCE QUOTE column [, ...]] | * ]
              [ESCAPE [AS] 'escape'] )]

          | 'CUSTOM' (Formatter=<formatter specifications>)
    [ ENCODING 'write_encoding' ]
    [ DISTRIBUTED BY ({column [opclass]}, [ ... ] ) | DISTRIBUTED RANDOMLY ]

CREATE WRITABLE EXTERNAL [TEMPORARY | TEMP] TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION('s3://S3_endpoint[:port]/bucket_name/[S3_prefix] [region=S3-
region] [config=config_file]')
      [ON MASTER]
      FORMAT 'TEXT'
              [( [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [ESCAPE [AS] 'escape' | 'OFF'] )]
         | 'CSV'
              [([QUOTE [AS] 'quote']
              [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [FORCE QUOTE column [, ...]] | * ]
              [ESCAPE [AS] 'escape'] )]

CREATE WRITABLE EXTERNAL WEB [TEMPORARY | TEMP] TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
    EXECUTE 'command' [ON ALL]
    FORMAT 'TEXT'
              [( [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [ESCAPE [AS] 'escape' | 'OFF'] )]
         | 'CSV'
              [([QUOTE [AS] 'quote']
              [DELIMITER [AS] 'delimiter']
              [NULL [AS] 'null string']
              [FORCE QUOTE column [, ...]] | * ]
              [ESCAPE [AS] 'escape'] )]
          | 'CUSTOM' (Formatter=<formatter specifications>)
    [ ENCODING 'write_encoding' ]
    [ DISTRIBUTED BY ({column [opclass]}, [ ... ] ) | DISTRIBUTED RANDOMLY ]
```

Description

`CREATE EXTERNAL TABLE` or `CREATE EXTERNAL WEB TABLE` creates a new readable external table definition in Greenplum Database. Readable external tables are typically used for fast, parallel data loading. Once an external table is defined, you can query its data directly (and in parallel) using SQL commands. For example, you can select, join, or sort external table data. You can also create views for external tables. DML operations (`UPDATE`, `INSERT`, `DELETE`, or `TRUNCATE`) are not allowed on readable external tables, and you cannot create indexes on readable external tables.

`CREATE WRITABLE EXTERNAL TABLE` or `CREATE WRITABLE EXTERNAL WEB TABLE` creates a new writable external table definition in Greenplum Database. Writable external tables are typically used for unloading data from the database into a set of files or named pipes. Writable external web tables can also be used to output data to an executable program. Writable external tables can also be used as output targets for Greenplum parallel MapReduce calculations. Once a writable external table is defined, data can

be selected from database tables and inserted into the writable external table. Writable external tables only allow `INSERT` operations – `SELECT`, `UPDATE`, `DELETE` or `TRUNCATE` are not allowed.

The main difference between regular external tables and external web tables is their data sources. Regular readable external tables access static flat files, whereas external web tables access dynamic data sources – either on a web server or by executing OS commands or scripts.

See *Working with External Data* for detailed information about working with external tables.

Parameters

`READABLE | WRITABLE`

Specifies the type of external table, readable being the default. Readable external tables are used for loading data into Greenplum Database. Writable external tables are used for unloading data.

`WEB`

Creates a readable or writable external web table definition in Greenplum Database. There are two forms of readable external web tables – those that access files via the `http://` protocol or those that access data by executing OS commands. Writable external web tables output data to an executable program that can accept an input stream of data. External web tables are not rescannable during query execution.

The `s3` protocol does not support external web tables. You can, however, create an external web table that executes a third-party tool to read data from or write data to S3 directly.

`TEMPORARY | TEMP`

If specified, creates a temporary readable or writable external table definition in Greenplum Database. Temporary external tables exist in a special schema; you cannot specify a schema name when you create the table. Temporary external tables are automatically dropped at the end of a session.

An existing permanent table with the same name is not visible to the current session while the temporary table exists, unless you reference the permanent table with its schema-qualified name.

`table_name`

The name of the new external table.

`column_name`

The name of a column to create in the external table definition. Unlike regular tables, external tables do not have column constraints or default values, so do not specify those.

`LIKE` ***`other_table`***

The `LIKE` clause specifies a table from which the new external table automatically copies all column names, data types and Greenplum distribution policy. If the original table specifies any column constraints or default column values, those will not be copied over to the new external table definition.

`data_type`

The data type of the column.

`LOCATION` ***`('protocol://[host[:port]]/path/file' [, ...])`***

If you use the `pxf` protocol to access an external data source, refer to *pxf:// Protocol* for information about the `pxf` protocol.

If you use the `s3` protocol to read or write to S3, refer to *About the S3 Protocol URL* for additional information about the `s3` protocol `LOCATION` clause syntax.

For readable external tables, specifies the URI of the external data source(s) to be used to populate the external table or web table. Regular readable external tables allow the

gpfdist or file protocols. External web tables allow the http protocol. If port is omitted, port 8080 is assumed for http and gpfdist protocols. If using the gpfdist protocol, the path is relative to the directory from which gpfdist is serving files (the directory specified when you started the gpfdist program). Also, gpfdist can use wildcards or other C-style pattern matching (for example, a whitespace character is [[:space:]]) to denote multiple files in a directory. For example:

```
'gpfdist://filehost:8081/*'
'gpfdist://masterhost/my_load_file'
'file://seghost1/dbfast1/external/myfile.txt'
'http://intranet.example.com/finance/expenses.csv'
```

For writable external tables, specifies the URI location of the gpfdist process or S3 protocol that will collect data output from the Greenplum segments and write it to one or more named files. For gpfdist the path is relative to the directory from which gpfdist is serving files (the directory specified when you started the gpfdist program). If multiple gpfdist locations are listed, the segments sending data will be evenly divided across the available output locations. For example:

```
'gpfdist://outputhost:8081/data1.out',
'gpfdist://outputhost:8081/data2.out'
```

With two gpfdist locations listed as in the above example, half of the segments would send their output data to the data1.out file and the other half to the data2.out file.

With the option #transform=*trans_name*, you can specify a transform to apply when loading or extracting data. The *trans_name* is the name of the transform in the YAML configuration file you specify with the you run the gpfdist utility. For information about specifying a transform, see *gpfdist* in the *Greenplum Utility Guide*.

ON MASTER

Restricts all table-related operations to the Greenplum master segment. Permitted only on readable and writable external tables created with the s3 or custom protocols. The gpfdist, gpfdists, pxf, and file protocols do not support ON MASTER.

> **Note:** Be aware of potential resource impacts when reading from or writing to external tables you create with the ON MASTER clause. You may encounter performance issues when you restrict table operations solely to the Greenplum master segment.

EXECUTE *'command'* [ON ...]

Allowed for readable external web tables or writable external tables only. For readable external web tables, specifies the OS command to be executed by the segment instances. The *command* can be a single OS command or a script. The ON clause is used to specify which segment instances will execute the given command.

- ON ALL is the default. The command will be executed by every active (primary) segment instance on all segment hosts in the Greenplum Database system. If the command executes a script, that script must reside in the same location on all of the segment hosts and be executable by the Greenplum superuser (gpadmin).
- ON MASTER runs the command on the master host only.

  > **Note:** Logging is not supported for external web tables when the ON MASTER clause is specified.
- ON *number* means the command will be executed by the specified number of segments. The particular segments are chosen randomly at runtime by the Greenplum Database system. If the command executes a script, that script must reside in the same location on all of the segment hosts and be executable by the Greenplum superuser (gpadmin).

- HOST means the command will be executed by one segment on each segment host (once per segment host), regardless of the number of active segment instances per host.
- HOST *segment_hostname* means the command will be executed by all active (primary) segment instances on the specified segment host.
- SEGMENT *segment_id* means the command will be executed only once by the specified segment. You can determine a segment instance's ID by looking at the *content* number in the system catalog table *gp_segment_configuration*. The *content* ID of the Greenplum Database master is always `-1`.

For writable external tables, the *command* specified in the `EXECUTE` clause must be prepared to have data piped into it. Since all segments that have data to send will write their output to the specified command or program, the only available option for the `ON` clause is `ON ALL`.

`FORMAT 'TEXT | CSV' (options)`

When the `FORMAT` clause identfies delimited text (`TEXT`) or comma separated values (`CSV`) format, formatting options are similar to those available with the PostgreSQL *`COPY`* command. If the data in the file does not use the default column delimiter, escape character, null string and so on, you must specify the additional formatting options so that the data in the external file is read correctly by Greenplum Database. For information about using a custom format, see "Loading and Unloading Data" in the *Greenplum Database Administrator Guide*.

If you use the `pxf` protocol to access an external data source, refer to *Accessing External Data with PXF* for information about using PXF.

`FORMAT 'CUSTOM' (formatter=formatter_specification)`

Specifies a custom data format. The *formatter_specification* specifies the function to use to format the data, followed by comma-separated parameters to the formatter function. The length of the formatter specification, the string including `Formatter=`, can be up to approximately 50K bytes.

If you use the `pxf` protocol to access an external data source, refer to *Accessing External Data with PXF* for information about using PXF.

For general information about using a custom format, see "Loading and Unloading Data" in the *Greenplum Database Administrator Guide*.

`DELIMITER`

Specifies a single ASCII character that separates columns within each row (line) of data. The default is a tab character in `TEXT` mode, a comma in `CSV` mode. In `TEXT` mode for readable external tables, the delimiter can be set to `OFF` for special use cases in which unstructured data is loaded into a single-column table.

For the `s3` protocol, the delimiter cannot be a newline character (`\n`) or a carriage return character (`\r`).

`NULL`

Specifies the string that represents a `NULL` value. The default is `\N` (backslash-N) in `TEXT` mode, and an empty value with no quotations in `CSV` mode. You might prefer an empty string even in `TEXT` mode for cases where you do not want to distinguish `NULL` values from empty strings. When using external and web tables, any data item that matches this string will be considered a `NULL` value.

As an example for the `text` format, this `FORMAT` clause can be used to specify that the string of two single quotes (`''`) is a `NULL` value.

```
FORMAT 'text' (delimiter ',' null '\'\'\'\'' )
```

`ESCAPE`

Specifies the single character that is used for C escape sequences (such as `\n`,`\t`,`\100`, and so on) and for escaping data characters that might otherwise be taken as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. The default escape character is a \ (backslash) for text-formatted files and a `"` (double quote) for csv-formatted files, however it is possible to specify another character to represent an escape. It is also possible to disable escaping in text-formatted files by specifying the value `'OFF'` as the escape value. This is very useful for data such as text-formatted web log data that has many embedded backslashes that are not intended to be escapes.

NEWLINE

Specifies the newline used in your data files – `LF` (Line feed, 0x0A), `CR` (Carriage return, 0x0D), or `CRLF` (Carriage return plus line feed, 0x0D 0x0A). If not specified, a Greenplum Database segment will detect the newline type by looking at the first row of data it receives and using the first newline type encountered.

HEADER

For readable external tables, specifies that the first line in the data file(s) is a header row (contains the names of the table columns) and should not be included as data for the table. If using multiple data source files, all files must have a header row.

For the `s3` protocol, the column names in the header row cannot contain a newline character (`\n`) or a carriage return (`\r`).

The `pxf` protocol does not support the `HEADER` formatting option.

QUOTE

Specifies the quotation character for `CSV` mode. The default is double-quote (`"`).

FORCE NOT NULL

In `CSV` mode, processes each specified column as though it were quoted and hence not a `NULL` value. For the default null string in `CSV` mode (nothing between two delimiters), this causes missing values to be evaluated as zero-length strings.

FORCE QUOTE

In `CSV` mode for writable external tables, forces quoting to be used for all non-`NULL` values in each specified column. If `*` is specified then non-`NULL` values will be quoted in all columns. `NULL` output is never quoted.

FILL MISSING FIELDS

In both `TEXT` and `CSV` mode for readable external tables, specifying `FILL MISSING FIELDS` will set missing trailing field values to `NULL` (instead of reporting an error) when a row of data has missing data fields at the end of a line or row. Blank rows, fields with a `NOT NULL` constraint, and trailing delimiters on a line will still report an error.

ENCODING *'encoding'*

Character set encoding to use for the external table. Specify a string constant (such as `'SQL_ASCII'`), an integer encoding number, or `DEFAULT` to use the default client encoding. See *Character Set Support*.

LOG ERRORS [PERSISTENTLY]

This is an optional clause that can precede a `SEGMENT REJECT LIMIT` clause to log information about rows with formatting errors. The error log data is stored internally. If error log data exists for a specified external table, new data is appended to existing error log data. The error log data is not replicated to mirror segments.

The data is deleted when the external table is dropped unless you specify the keyword `PERSISTENTLY`. If the keyword is specified, the log data persists after the external table is dropped.

The error log data is accessed with the Greenplum Database built-in SQL function `gp_read_error_log()`, or with the SQL function `gp_read_persistent_error_log()` if the `PERSISTENTLY` keyword is specified.

If you use the `PERSISTENTLY` keyword, you must install the functions that manage the persistent error log information.

See *Notes* for information about the error log information and built-in functions for viewing and managing error log information.

`SEGMENT REJECT LIMIT` *`count`* `[ROWS | PERCENT]`

Runs a `COPY FROM` operation in single row error isolation mode. If the input rows have format errors they will be discarded provided that the reject limit count is not reached on any Greenplum segment instance during the load operation. The reject limit count can be specified as number of rows (the default) or percentage of total rows (1-100). If `PERCENT` is used, each segment starts calculating the bad row percentage only after the number of rows specified by the parameter `gp_reject_percent_threshold` has been processed. The default for `gp_reject_percent_threshold` is 300 rows. Constraint errors such as violation of a `NOT NULL`, `CHECK`, or `UNIQUE` constraint will still be handled in "all-or-nothing" input mode. If the limit is not reached, all good rows will be loaded and any error rows discarded.

Note: When reading an external table, Greenplum Database limits the initial number of rows that can contain formatting errors if the `SEGMENT REJECT LIMIT` is not triggered first or is not specified. If the first 1000 rows are rejected, the `COPY` operation is stopped and rolled back.

The limit for the number of initial rejected rows can be changed with the Greenplum Database server configuration parameter `gp_initial_bad_row_limit`. See *Server Configuration Parameters* for information about the parameter.

`DISTRIBUTED BY ({`*`column`* `[`*`opclass`*`]}, [ ... ] )`
`DISTRIBUTED RANDOMLY`

Used to declare the Greenplum Database distribution policy for a writable external table. By default, writable external tables are distributed randomly. If the source table you are exporting data from has a hash distribution policy, defining the same distribution key column(s) and operator class(es), `oplcass`, for the writable external table will improve unload performance by eliminating the need to move rows over the interconnect. When you issue an unload command such as `INSERT INTO` *`wex_table`* `SELECT * FROM` *`source_table`*, the rows that are unloaded can be sent directly from the segments to the output location if the two tables have the same hash distribution policy.

Examples

Start the `gpfdist` file server program in the background on port `8081` serving files from directory `/var/data/staging`:

```
gpfdist -p 8081 -d /var/data/staging -l /home/gpadmin/log &
```

Create a readable external table named `ext_customer` using the `gpfdist` protocol and any text formatted files (`*.txt`) found in the `gpfdist` directory. The files are formatted with a pipe (`|`) as the column delimiter and an empty space as `NULL`. Also access the external table in single row error isolation mode:

```
CREATE EXTERNAL TABLE ext_customer
   (id int, name text, sponsor text)
   LOCATION ( 'gpfdist://filehost:8081/*.txt' )
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
```

```
   LOG ERRORS SEGMENT REJECT LIMIT 5;
```

Create the same readable external table definition as above, but with CSV formatted files:

```
CREATE EXTERNAL TABLE ext_customer
   (id int, name text, sponsor text)
   LOCATION ( 'gpfdist://filehost:8081/*.csv' )
   FORMAT 'CSV' ( DELIMITER ',' );
```

Create a readable external table named `ext_expenses` using the `file` protocol and several CSV formatted files that have a header row:

```
CREATE EXTERNAL TABLE ext_expenses (name text, date date,
amount float4, category text, description text)
LOCATION (
'file://seghost1/dbfast/external/expenses1.csv',
'file://seghost1/dbfast/external/expenses2.csv',
'file://seghost2/dbfast/external/expenses3.csv',
'file://seghost2/dbfast/external/expenses4.csv',
'file://seghost3/dbfast/external/expenses5.csv',
'file://seghost3/dbfast/external/expenses6.csv'
)
FORMAT 'CSV' ( HEADER );
```

Create a readable external web table that executes a script once per segment host:

```
CREATE EXTERNAL WEB TABLE log_output (linenum int, message
text)  EXECUTE '/var/load_scripts/get_log_data.sh' ON HOST
 FORMAT 'TEXT' (DELIMITER '|');
```

Create a writable external table named `sales_out` that uses `gpfdist` to write output data to a file named `sales.out`. The files are formatted with a pipe (`|`) as the column delimiter and an empty space as `NULL`.

```
CREATE WRITABLE EXTERNAL TABLE sales_out (LIKE sales)
   LOCATION ('gpfdist://etl1:8081/sales.out')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
   DISTRIBUTED BY (txn_id);
```

Create a writable external web table that pipes output data received by the segments to an executable script named `to_adreport_etl.sh`:

```
CREATE WRITABLE EXTERNAL WEB TABLE campaign_out
(LIKE campaign)
 EXECUTE '/var/unload_scripts/to_adreport_etl.sh'
 FORMAT 'TEXT' (DELIMITER '|');
```

Use the writable external table defined above to unload selected data:

```
INSERT INTO campaign_out SELECT * FROM campaign WHERE
customer_id=123;
```

Notes

When you specify the `LOG ERRORS` clause, Greenplum Database captures errors that occur while reading the external table data. For information about the error log format, see *Viewing Bad Rows in the Error Log*.

You can view and manage the captured error log data. The functions to manage log data depend on whether the data is persistent (the `PERSISTENTLY` keyword is used with the `LOG ERRORS` clause).

- Functions that manage non-persistent error log data from external tables that were defined without the PERSISTENTLY keyword.
  - The built-in SQL function gp_read_error_log('*table_name*') displays error log information for an external table. This example displays the error log data from the external table ext_expenses.

    ```
    SELECT * from gp_read_error_log('ext_expenses');
    ```

    The function returns no data if you created the external table with the LOG ERRORS PERSISTENTLY clause, or if the external table does not exist.
  - The built-in SQL function gp_truncate_error_log('*table_name*') deletes the error log data for *table_name*. This example deletes the error log data captured from the external table ext_expenses:

    ```
    SELECT gp_truncate_error_log('ext_expenses');
    ```

    Dropping the table also deletes the table's log data. The function does not truncate log data if the external table is defined with the LOG ERRORS PERSISTENTLY clause.

    The function returns FALSE if the table does not exist.
- Functions that manage persistent error log data from external tables that were defined with the PERSISTENTLY keyword.

  Note: The functions that manage persistent error log data from external tables are defined in the file $GPHOME/share/postgresql/contrib/gpexterrorhandle.sql. The functions must be installed in the databases that use persistent error log data from an external table. This psql command installs the functions into the database testdb.

  ```
  psql -d test -U gpadmin -f $GPHOME/share/postgresql/contrib/
  gpexterrorhandle.sql
  ```

  - The SQL function gp_read_persistent_error_log('*table_name*') displays persistent log data for an external table.

    The function returns no data if you created the external table without the PERSISTENTLY keyword. The function returns persistent log data for an external table even after the table has been dropped.
  - The SQL function gp_truncate_persistent_error_log('*table_name*') truncates persistent log data for a table.

    For persistent log data, you must manually delete the data. Dropping the external table does not delete persistent log data.
- These items apply to both non-persistent and persistent error log data and the related functions.
  - The gp_read_* functions require SELECT privilege on the table.
  - The gp_truncate_* functions require owner privilege on the table.
  - You can use the * wildcard character to delete error log information for existing tables in the current database. Specify the string *.* to delete all database error log information, including error log information that was not deleted due to previous database issues. If * is specified, database owner privilege is required. If *.* is specified, operating system super-user privilege is required. Non-persistent and persistent error log data must be deleted with their respective gp_truncate_* functions.

When multiple Greenplum Database external tables are defined with the gpfdist, gpfdists, or file protocol and access the same named pipe a Linux system, Greenplum Database restricts access to the named pipe to a single reader. An error is returned if a second reader attempts to access the named pipe.

Compatibility

CREATE EXTERNAL TABLE is a Greenplum Database extension. The SQL standard makes no provisions for external tables.

See Also

CREATE TABLE AS, *CREATE TABLE*, *COPY*, *SELECT INTO*, *INSERT*

CREATE FOREIGN DATA WRAPPER

Defines a new foreign-data wrapper.

Synopsis

```
CREATE FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( [ mpp_execute { 'master' | 'any' | 'all segments' }
  [, ] ] option 'value' [, ... ] ) ]
```

Description

`CREATE FOREIGN DATA WRAPPER` creates a new foreign-data wrapper in the current database. The user who defines the foreign-data wrapper becomes its owner.

Only superusers can create foreign-data wrappers.

Parameters

name

The name of the foreign-data wrapper to create. The *name* must be unique within the database.

`HANDLER` *`handler_function`*

The name of a previously registered function that Greenplum Database calls to retrieve the execution functions for foreign tables. *hander_function* must take no arguments, and its return type must be `fdw_handler`.

It is possible to create a foreign-data wrapper with no handler function, but you can only declare, not access, foreign tables using such a wrapper.

`VALIDATOR` *`validator_function`*

The name of a previously registered function that Greenplum Database calls to check the options provided to the foreign-data wrapper. This function also checks the options for foreign servers, user mappings, and foreign tables that use the foreign-data wrapper. If no validator function or `NO VALIDATOR` is specified, Greenplum Database does not check options at creation time. (Depending upon the implementation, foreign-data wrappers may ignore or reject invalid options at runtime.)

validator_function must take two arguments: one of type `text[]`, which contains the array of options as stored in the system catalogs, and one of type `oid`, which identifies the OID of the system catalog containing the options.

The return type is ignored; *validator_function* should report invalid options using the `ereport(ERROR)` function.

`OPTIONS ( `*`option`* **`'`*`value`***`' [, ... ] )`**

The options for the new foreign-data wrapper. Option names must be unique. The option names and values are foreign-data wrapper-specific and are validated using the foreign-data wrappers' *validator_function*.

`mpp_execute { 'master' | 'any' | 'all segments' }`

An option that identifies the host from which the foreign data-wrapper requests data:

- `master` (the default)—Request data from the master host.
- `any`—Request data from either the master host or any one segment, depending on which path costs less.
- `all segments`—Request data from all segments. To support this option value, the foreign data-wrapper must have a policy that matches the segments to data.

The `mpp_execute` option can be specified in multiple commands: `CREATE FOREIGN TABLE`, `CREATE SERVER`, and `CREATE FOREIGN DATA WRAPPER`. The foreign table setting takes precedence over the foreign server setting, followed by the foreign data wrapper setting.

Notes

The foreign-data wrapper functionality is still under development. Optimization of queries is primitive (and mostly left to the wrapper).

Examples

Create a useless foreign-data wrapper named `dummy`:

```
CREATE FOREIGN DATA WRAPPER dummy;
```

Create a foreign-data wrapper named `file` with a handler function named `file_fdw_handler`:

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

Create a foreign-data wrapper named `mywrapper` that includes an option:

```
CREATE FOREIGN DATA WRAPPER mywrapper OPTIONS (debug 'true');
```

Compatibility

`CREATE FOREIGN DATA WRAPPER` conforms to ISO/IEC 9075-9 (SQL/MED), with the exception that the `HANDLER` and `VALIDATOR` clauses are extensions, and the standard clauses `LIBRARY` and `LANGUAGE` are not implemented in Greenplum Database.

Note, however, that the SQL/MED functionality as a whole is not yet conforming.

See Also

`ALTER FOREIGN DATA WRAPPER`, *`DROP FOREIGN DATA WRAPPER`*, *`CREATE SERVER`*, *`CREATE USER MAPPING`*

CREATE FOREIGN TABLE

Defines a new foreign table.

Synopsis

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [
    column_name data_type [ OPTIONS ( option 'value' [, ... ] ) ]
 [ COLLATE collation ] [ column_constraint [ ... ] ]
      [, ... ]
] )
    SERVER server_name
  [ OPTIONS ( [ mpp_execute { 'master' | 'any' | 'all segments' }
 [, ] ] option 'value' [, ... ] ) ]
```

where *column_constraint* is:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  DEFAULT default_expr }
```

Description

`CREATE FOREIGN TABLE` creates a new foreign table in the current database. The user who creates the foreign table becomes its owner.

If you schema-qualify the table name (for example, `CREATE FOREIGN TABLE myschema.mytable ...`), Greenplum Database creates the table in the specified schema. Otherwise, the foreign table is created in the current schema. The name of the foreign table must be distinct from the name of any other foreign table, table, sequence, index, or view in the same schema.

Because `CREATE FOREIGN TABLE` automatically creates a data type that represents the composite type corresponding to one row of the foreign table, foreign tables cannot have the same name as any existing data type in the same schema.

To create a foreign table, you must have `USAGE` privilege on the foreign server, as well as `USAGE` privilege on all column types used in the table.

Parameters

`IF NOT EXISTS`

Do not throw an error if a relation with the same name already exists. Greenplum Database issues a notice in this case. Note that there is no guarantee that the existing relation is anything like the one that would have been created.

`table_name`

The name (optionally schema-qualified) of the foreign table to create.

`column_name`

The name of a column to create in the new foreign table.

`data_type`

The data type of the column, including array specifiers.

`NOT NULL`

The column is not allowed to contain null values.

`NULL`

The column is allowed to contain null values. This is the default.

This clause is provided only for compatibility with non-standard SQL databases. Its use is discouraged in new applications.

`DEFAULT` ***`default_expr`***

The `DEFAULT` clause assigns a default value for the column whose definition it appears within. The value is any variable-free expression; Greenplum Database does not allow subqueries and cross-references to other columns in the current table. The data type of the default expression must match the data type of the column.

Greenplum Database uses the default expression in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

`server_name`

The name of an existing server to use for the foreign table. For details on defining a server, see *CREATE SERVER*.

OPTIONS (*option* '*value*' [, ...])

The options for the new foreign table or one of its columns. While option names must be unique, a table option and a column option may have the same name. The option names and values are foreign-data wrapper-specific. Greenplum Database validates the options and values using the foreign-data wrapper's *validator_function*.

mpp_execute { 'master' | 'any' | 'all segments' }

An option that identifies the host from which the foreign data-wrapper requests data:

- `master` (the default)—Request data from the master host.
- `any`—Request data from either the master host or any one segment, depending on which path costs less.
- `all segments`—Request data from all segments. To support this option value, the foreign data-wrapper must have a policy that matches the segments to data.

Use of the foreign table `mpp_execute` option, and the specific modes supported, is foreign data-wrapper-specific.

The `mpp_execute` option can be specified in multiple commands: `CREATE FOREIGN TABLE`, `CREATE SERVER`, and `CREATE FOREIGN DATA WRAPPER`. The foreign table setting takes precedence over the foreign server setting, followed by the foreign data wrapper setting.

Notes

The Pivotal Query Optimizer, GPORCA, does not support foreign tables. A query on a foreign table always falls back to the Postgres Planner.

Examples

Create a foreign table named `films` with the server named `film_server`:

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
)
SERVER film_server;
```

Compatibility

`CREATE FOREIGN TABLE` largely conforms to the SQL standard; however, much as with *CREATE TABLE*, Greenplum Database permits `NULL` constraints and zero-column foreign tables. The ability to specify a default value is a Greenplum Database extension, as is the `mpp_execute` option.

See Also

ALTER FOREIGN TABLE, *DROP FOREIGN TABLE*, *CREATE TABLE*, *CREATE SERVER*

CREATE FUNCTION

Defines a new function.

Synopsis

```
CREATE [OR REPLACE] FUNCTION name
    ( [ [argmode] [argname] argtype [ { DEFAULT | = } default_expr ]
 [, ...] ] )
      [ RETURNS rettype
        | RETURNS TABLE ( column_name column_type [, ...] ) ]
    { LANGUAGE langname
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [NOT] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    | EXECUTE ON { ANY | MASTER | ALL SEGMENTS | INITPLAN }
    | COST execution_cost
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol' } ...
    [ WITH ({ DESCRIBE = describe_function
           } [, ...] ) ]
```

Description

`CREATE FUNCTION` defines a new function. `CREATE OR REPLACE FUNCTION` either creates a new function, or replaces an existing definition.

The name of the new function must not match any existing function with the same input argument types in the same schema. However, functions of different argument types may share a name (overloading).

To update the definition of an existing function, use `CREATE OR REPLACE FUNCTION`. It is not possible to change the name or argument types of a function this way (this would actually create a new, distinct function). Also, `CREATE OR REPLACE FUNCTION` will not let you change the return type of an existing function. To do that, you must drop and recreate the function. When using `OUT` parameters, that means you cannot change the types of any `OUT` parameters except by dropping the function. If you drop and then recreate a function, you will have to drop existing objects (rules, views, triggers, and so on) that refer to the old function. Use `CREATE OR REPLACE FUNCTION` to change a function definition without breaking objects that refer to the function.

The user that creates the function becomes the owner of the function.

To be able to create a function, you must have `USAGE` privilege on the argument types and the return type.

For more information about creating functions, see the *User Defined Functions* section of the PostgreSQL documentation.

Limited Use of VOLATILE and STABLE Functions

To prevent data from becoming out-of-sync across the segments in Greenplum Database, any function classified as `STABLE` or `VOLATILE` cannot be executed at the segment level if it contains SQL or modifies the database in any way. For example, functions such as `random()` or `timeofday()` are not allowed to execute on distributed data in Greenplum Database because they could potentially cause inconsistent data between the segment instances.

To ensure data consistency, `VOLATILE` and `STABLE` functions can safely be used in statements that are evaluated on and execute from the master. For example, the following statements are always executed on the master (statements without a `FROM` clause):

```
SELECT setval('myseq', 201);
SELECT foo();
```

In cases where a statement has a FROM clause containing a distributed table and the function used in the FROM clause simply returns a set of rows, execution may be allowed on the segments:

```
SELECT * FROM foo();
```

One exception to this rule are functions that return a table reference (rangeFuncs) or functions that use the refCursor data type. Note that you cannot return a refcursor from any kind of function in Greenplum Database.

Function Volatility and EXECUTE ON Attributes

Volatility attributes (IMMUTABLE, STABLE, VOLATILE) and EXECUTE ON attributes specify two different aspects of function execution. In general, volatility indicates when the function is executed, and EXECUTE ON indicates where it is executed.

For example, a function defined with the IMMUTABLE attribute can be executed at query planning time, while a function with the VOLATILE attribute must be executed for every row in the query. A function with the EXECUTE ON MASTER attribute is executed only on the master segment and a function with the EXECUTE ON ALL SEGMENTS attribute is executed on all primary segment instances (not the master).

See *Using Functions and Operators* in the *Greenplum Database Administrator Guide*.

Functions And Replicated Tables

A user-defined function that executes only SELECT commands on replicated tables can run on segments. Replicated tables, created with the DISTRIBUTED REPLICATED clause, store all of their rows on every segment. It is safe for a function to read them on the segments, but updates to replicated tables must execute on the master instance.

Parameters

name

The name (optionally schema-qualified) of the function to create.

argmode

The mode of an argument: either IN, OUT, INOUT, or VARIADIC. If omitted, the default is IN. Only OUT arguments can follow an argument declared as VARIADIC. Also, OUT and INOUT arguments cannot be used together with the RETURNS TABLE notation.

argname

The name of an argument. Some languages (currently only SQL and PL/pgSQL) let you use the name in the function body. For other languages the name of an input argument is just extra documentation, so far as the function itself is concerned; but you can use input argument names when calling a function to improve readability. In any case, the name of an output argument is significant, since it defines the column name in the result row type. (If you omit the name for an output argument, the system will choose a default column name.)

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any. The argument types may be base, composite, or domain types, or may reference the type of a table column.

Depending on the implementation language it may also be allowed to specify pseudotypes such as cstring. Pseudotypes indicate that the actual argument type is either incompletely specified, or outside the set of ordinary SQL data types.

The type of a column is referenced by writing *tablename.columnname*%TYPE. Using this feature can sometimes help make a function independent of changes to the definition of a table.

default_expr

An expression to be used as the default value if the parameter is not specified. The expression must be coercible to the argument type of the parameter. Only `IN` and `INOUT` parameters can have a default value. Each input parameter in the argument list that follows a parameter with a default value must have a default value as well.

rettype

The return data type (optionally schema-qualified). The return type can be a base, composite, or domain type, or may reference the type of a table column. Depending on the implementation language it may also be allowed to specify pseudotypes such as `cstring`. If the function is not supposed to return a value, specify `void` as the return type.

When there are `OUT` or `INOUT` parameters, the `RETURNS` clause may be omitted. If present, it must agree with the result type implied by the output parameters: `RECORD` if there are multiple output parameters, or the same type as the single output parameter.

The `SETOF` modifier indicates that the function will return a set of items, rather than a single item.

The type of a column is referenced by writing *`tablename.columnname`*`%TYPE`.

column_name

The name of an output column in the `RETURNS TABLE` syntax. This is effectively another way of declaring a named `OUT` parameter, except that `RETURNS TABLE` also implies `RETURNS SETOF`.

column_type

The data type of an output column in the `RETURNS TABLE` syntax.

langname

The name of the language that the function is implemented in. May be `SQL`, `C`, `internal`, or the name of a user-defined procedural language. See *`CREATE LANGUAGE`* for the procedural languages supported in Greenplum Database. For backward compatibility, the name may be enclosed by single quotes.

WINDOW

`WINDOW` indicates that the function is a window function rather than a plain function. This is currently only useful for functions written in C. The `WINDOW` attribute cannot be changed when replacing an existing function definition.

IMMUTABLE
STABLE
VOLATILE
LEAKPROOF

These attributes inform the query optimizer about the behavior of the function. At most one choice may be specified. If none of these appear, `VOLATILE` is the default assumption. Since Greenplum Database currently has limited use of `VOLATILE` functions, if a function is truly `IMMUTABLE`, you must declare it as so to be able to use it without restrictions.

`IMMUTABLE` indicates that the function cannot modify the database and always returns the same result when given the same argument values. It does not do database lookups or otherwise use information not directly present in its argument list. If this option is given, any call of the function with all-constant arguments can be immediately replaced with the function value.

`STABLE` indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for functions whose results depend on database lookups, parameter values (such as the current time zone), and so on. Also note that the *current_timestamp* family of functions qualify as stable, since their values do not change within a transaction.

VOLATILE indicates that the function value can change even within a single table scan, so no optimizations can be made. Relatively few database functions are volatile in this sense; some examples are random(), timeofday(). But note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away; an example is setval().

LEAKPROOF indicates that the function has no side effects. It reveals no information about its arguments other than by its return value. For example, a function that throws an error message for some argument values but not others, or that includes the argument values in any error message, is not leakproof. The query planner may push leakproof functions (but not others) into views created with the security_barrier option. See *CREATE VIEW* and *CREATE RULE*. This option can only be set by the superuser.

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT (the default) indicates that the function will be called normally when some of its arguments are null. It is then the function author's responsibility to check for null values if necessary and respond appropriately. RETURNS NULL ON NULL INPUT or STRICT indicates that the function always returns null whenever any of its arguments are null. If this parameter is specified, the function is not executed when there are null arguments; instead a null result is assumed automatically.

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER (the default) indicates that the function is to be executed with the privileges of the user that calls it. SECURITY DEFINER specifies that the function is to be executed with the privileges of the user that created it. The key word EXTERNAL is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all functions not just external ones.

EXECUTE ON ANY
EXECUTE ON MASTER
EXECUTE ON ALL SEGMENTS
EXECUTE ON INITPLAN

The EXECUTE ON attributes specify where (master or segment instance) a function executes when it is invoked during the query execution process.

EXECUTE ON ANY (the default) indicates that the function can be executed on the master, or any segment instance, and it returns the same result regardless of where it is executed. Greenplum Database determines where the function executes.

EXECUTE ON MASTER indicates that the function must execute only on the master instance.

EXECUTE ON ALL SEGMENTS indicates that the function must execute on all primary segment instances, but not the master, for each invocation. The overall result of the function is the UNION ALL of the results from all segment instances.

EXECUTE ON INITPLAN indicates that the function contains an SQL command that dispatches queries to the segment instances and requires special processing on the master instance by Greenplum Database when possible.

> **Note:** EXECUTE ON INITPLAN is only supported in functions that are used in the FROM clause of a CREATE TABLE AS or INSERT command such as the get_data() function in these commands.

```
CREATE TABLE t AS SELECT * FROM get_data();

INSERT INTO t1 SELECT * FROM get_data();
```

Greenplum Database does not support the `EXECUTE ON INITPLAN` attribute in a function that is used in the `WITH` clause of a query, a CTE (common table expression). For example, specifying `EXECUTE ON INITPLAN` in function `get_data()` in this CTE is not supported.

```
WITH tbl_a AS (SELECT * FROM get_data() )
   SELECT * from tbl_a
   UNION
   SELECT * FROM tbl_b;
```

For information about using `EXECUTE ON` attributes, see *Notes*.

`COST` *`execution_cost`*

A positive number identifying the estimated execution cost for the function, in *cpu_operator_cost* units. If the function returns a set, *execution_cost* identifies the cost per returned row. If the cost is not specified, C-language and internal functions default to 1 unit, while functions in other languages default to 100 units. The planner tries to evaluate the function less often when you specify larger *execution_cost* values.

`configuration_parameter`
`value`

The `SET` clause applies a value to a session configuration parameter when the function is entered. The configuration parameter is restored to its prior value when the function exits. `SET FROM CURRENT` saves the value of the parameter that is current when `CREATE FUNCTION` is executed as the value to be applied when the function is entered.

`definition`

A string constant defining the function; the meaning depends on the language. It may be an internal function name, the path to an object file, an SQL command, or text in a procedural language.

`obj_file, link_symbol`

This form of the `AS` clause is used for dynamically loadable C language functions when the function name in the C language source code is not the same as the name of the SQL function. The string *obj_file* is the name of the file containing the dynamically loadable object, and *link_symbol* is the name of the function in the C language source code. If the link symbol is omitted, it is assumed to be the same as the name of the SQL function being defined. The C names of all functions must be different, so you must give overloaded SQL functions different C names (for example, use the argument types as part of the C names). It is recommended to locate shared libraries either relative to `$libdir` (which is located at `$GPHOME/lib`) or through the dynamic library path (set by the `dynamic_library_path` server configuration parameter). This simplifies version upgrades if the new installation is at a different location.

`describe_function`

The name of a callback function to execute when a query that calls this function is parsed. The callback function returns a tuple descriptor that indicates the result type.

Notes

Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files. It is recommended to locate shared libraries either relative to `$libdir` (which is located at `$GPHOME/lib`) or through the dynamic library path (set by the `dynamic_library_path` server configuration parameter) on all master segment instances in the Greenplum array.

The full SQL type syntax is allowed for input arguments and return value. However, some details of the type specification (such as the precision field for type *numeric*) are the responsibility of the underlying function implementation and are not recognized or enforced by the `CREATE FUNCTION` command.

Greenplum Database allows function overloading. The same name can be used for several different functions so long as they have distinct input argument types. However, the C names of all functions must be different, so you must give overloaded C functions different C names (for example, use the argument types as part of the C names).

Two functions are considered the same if they have the same names and input argument types, ignoring any `OUT` parameters. Thus for example these declarations conflict:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

Functions that have different argument type lists are not considered to conflict at creation time, but if argument defaults are provided, they might conflict in use. For example, consider:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

The call `foo(10)`, will fail due to the ambiguity about which function should be called.

When repeated `CREATE FUNCTION` calls refer to the same object file, the file is only loaded once. To unload and reload the file, use the `LOAD` command.

You must have the `USAGE` privilege on a language to be able to define a function using that language.

It is often helpful to use dollar quoting to write the function definition string, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the function definition must be escaped by doubling them. A dollar-quoted string constant consists of a dollar sign ($), an optional tag of zero or more characters, another dollar sign, an arbitrary sequence of characters that makes up the string content, a dollar sign, the same tag that began this dollar quote, and a dollar sign. Inside the dollar-quoted string, single quotes, backslashes, or any character can be used without escaping. The string content is always written literally. For example, here are two different ways to specify the string "Dianne's horse" using dollar quoting:

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

If a `SET` clause is attached to a function, the effects of a `SET LOCAL` command executed inside the function for the same variable are restricted to the function; the configuration parameter's prior value is still restored when the function exits. However, an ordinary `SET` command (without `LOCAL`) overrides the `CREATE FUNCTION SET` clause, much as it would for a previous `SET LOCAL` command. The effects of such a command will persist after the function exits, unless the current transaction is rolled back.

If a function with a `VARIADIC` argument is declared as `STRICT`, the strictness check tests that the variadic array as a whole is non-null. PL/pgSQL will still call the function if the array has null elements.

When replacing an existing function with `CREATE OR REPLACE FUNCTION`, there are restrictions on changing parameter names. You cannot change the name already assigned to any input parameter (although you can add names to parameters that had none before). If there is more than one output parameter, you cannot change the names of the output parameters, because that would change the column names of the anonymous composite type that describes the function's result. These restrictions are made to ensure that existing calls of the function do not stop working when it is replaced.

Using Functions with Queries on Distributed Data

In some cases, Greenplum Database does not support using functions in a query where the data in a table specified in the FROM clause is distributed over Greenplum Database segments. As an example, this SQL query contains the function `func()`:

```
SELECT func(a) FROM table1;
```

The function is not supported for use in the query if all of the following conditions are met:

- The data of table `table1` is distributed over Greenplum Database segments.
- The function `func()` reads or modifies data from distributed tables.
- The function `func()` returns more than one row or takes an argument (a) that comes from `table1`.

If any of the conditions are not met, the function is supported. Specifically, the function is supported if any of the following conditions apply:

- The function `func()` does not access data from distributed tables, or accesses data that is only on the Greenplum Database master.
- The table `table1` is a master only table.
- The function `func()` returns only one row and only takes input arguments that are constant values. The function is supported if it can be changed to require no input arguments.

Using EXECUTE ON attributes

Most functions that execute queries to access tables can only execute on the master. However, functions that execute only SELECT queries on replicated tables can run on segments. If the function accesses a hash-distributed table or a randomly distributed table, the function should be defined with the EXECUTE ON MASTER attribute. Otherwise, the function might return incorrect results when the function is used in a complicated query. Without the attribute, planner optimization might determine it would be beneficial to push the function invocation to segment instances.

These are limitations for functions defined with the EXECUTE ON MASTER or EXECUTE ON ALL SEGMENTS attribute:

- The function must be a set-returning function.
- The function cannot be in the FROM clause of a query.
- The function cannot be in the SELECT list of a query with a FROM clause.
- A query that includes the function falls back from GPORCA to the Postgres Planner.

The attribute EXECUTE ON INITPLAN indicates that the function contains an SQL command that dispatches queries to the segment instances and requires special processing on the master instance by Greenplum Database. When possible, Greenplum Database handles the function on the master instance in the following manner.

1. First, Greenplum Database executes the function as part of an InitPlan node on the master instance and holds the function output temporarily.
2. Then, in the MainPlan of the query plan, the function is called in an EntryDB (a special query executor (QE) that runs on the master instance) and Greenplum Database returns the data that was captured when the function was executed as part of the InitPlan node. The function is not executed in the MainPlan.

This simple example uses the function `get_data()` in a CTAS command to create a table using data from the table `country`. The function contains a SELECT command that retrieves data from the table `country` and uses the EXECUTE ON INITPLAN attribute.

```
CREATE TABLE country(
  c_id integer, c_name text, region int)
  DISTRIBUTED RANDOMLY;

INSERT INTO country VALUES (11,'INDIA', 1 ), (22,'CANADA', 2), (33,'USA',
 3);
```

```
CREATE OR REPLACE FUNCTION get_data()
  RETURNS TABLE (
   c_id integer, c_name text
   )
AS $$
  SELECT
    c.c_id, c.c_name
  FROM
    country c;
$$
LANGUAGE SQL EXECUTE ON INITPLAN;

CREATE TABLE t AS SELECT * FROM get_data() DISTRIBUTED RANDOMLY;
```

If you view the query plan of the CTAS command with `EXPLAIN ANALYZE VERBOSE`, the plan shows that the function is run as part of an InitPlan node, and one of the listed slices is labeled as `entry db`. The query plan of a simple CTAS command without the function does not have an InitPlan node or an `entry db` slice.

If the function did not contain the `EXECUTE ON INITPLAN` attribute, the CTAS command returns the error `function cannot execute on a QE slice`.

When a function uses the `EXECUTE ON INITPLAN` attribute, a command that uses the function such as `CREATE TABLE t AS SELECT * FROM get_data()` gathers the results of the function onto the master segment and then redistributes the results to segment instances when inserting the data. If the function returns a large amount of data, the master might become a bottleneck when gathering and redistributing data. Performance might improve if you rewrite the function to run the CTAS command in the user defined function and use the table name as an input parameter. In this example, the function executes a CTAS command and does not require the `EXECUTE ON INITPLAN` attribute. Running the `SELECT` command creates the table `t1` using the function that executes the CTAS command.

```
CREATE OR REPLACE FUNCTION my_ctas(_tbl text) RETURNS VOID AS
$$
BEGIN
  EXECUTE format('CREATE TABLE %s AS SELECT c.c_id, c.c_name FROM country c
 DISTRIBUTED RANDOMLY', _tbl);
END
$$
LANGUAGE plpgsql;

SELECT my_ctas('t1');
```

Examples

A very simple addition function:

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

Increment an integer, making use of an argument name, in PL/pgSQL:

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS
integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

Increase the default segment host memory per query for a PL/pgSQL function:

```
CREATE OR REPLACE FUNCTION function_with_query() RETURNS
SETOF text AS $$
        BEGIN
                RETURN QUERY
                EXPLAIN ANALYZE SELECT * FROM large_table;
        END;
$$ LANGUAGE plpgsql
SET statement_mem='256MB';
```

Use polymorphic types to return an `ENUM` array:

```
CREATE TYPE rainbow AS
 ENUM('red','orange','yellow','green','blue','indigo','violet');
CREATE FUNCTION return_enum_as_array( anyenum, anyelement, anyelement )
    RETURNS TABLE (ae anyenum, aa anyarray) AS $$
    SELECT $1, array[$2, $3]
$$ LANGUAGE SQL STABLE;

SELECT * FROM return_enum_as_array('red'::rainbow, 'green'::rainbow,
 'blue'::rainbow);
```

Return a record containing multiple output parameters:

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

You can do the same thing more verbosely with an explicitly named composite type:

```
CREATE TYPE dup_result AS (f1 int, f2 text);
CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Another way to return multiple columns is to use a `TABLE` function:

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(4);
```

This function is defined with the `EXECUTE ON ALL SEGMENTS` to run on all primary segment instances. The `SELECT` command executes the function that returns the time it was run on each segment instance.

```
CREATE FUNCTION run_on_segs (text) returns setof text as $$
  begin
    return next ($1 || ' - ' || now()::text );
  end;
 $$ language plpgsql VOLATILE EXECUTE ON ALL SEGMENTS;

SELECT run_on_segs('my test');
```

This function looks up a part name in the parts table. The parts table is replicated, so the function can execute on the master or on the primary segments.

```
CREATE OR REPLACE FUNCTION get_part_name(partno int) RETURNS text AS
$$
DECLARE
   result text := ' ';
BEGIN
    SELECT part_name INTO result FROM parts WHERE part_id = partno;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

If you execute `SELECT get_part_name(100);` at the master the function executes on the master. (The master instance directs the query to a single primary segment.) If orders is a distributed table and you execute the following query, the `get_part_name()` function executes on the primary segments.

```
SELECT order_id, get_part_name(orders.part_no) FROM orders;
```

Compatibility

`CREATE FUNCTION` is defined in SQL:1999 and later. The Greenplum Database version is similar but not fully compatible. The attributes are not portable, neither are the different available languages.

For compatibility with some other database systems, *argmode* can be written either before or after *argname*. But only the first way is standard-compliant.

For parameter defaults, the SQL standard specifies only the syntax with the `DEFAULT` key word. The syntax with = is used in T-SQL and Firebird.

See Also

ALTER FUNCTION, *DROP FUNCTION*, *LOAD*

CREATE GROUP

Defines a new database role.

Synopsis

```
CREATE GROUP name [[WITH] option [ ... ]]
```

where *option* can be:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | CREATEEXTTABLE | NOCREATEEXTTABLE
      [ ( attribute='value'[, ...] ) ]
           where attributes and value are:
           type='readable'|'writable'
           protocol='gpfdist'|'http'
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | ROLE rolename [, ...]
```

```
    | ADMIN rolename [, ...]
    | RESOURCE QUEUE queue_name
    | RESOURCE GROUP group_name
    | [ DENY deny_point ]
    | [ DENY BETWEEN deny_point AND deny_point]
```

Description

`CREATE GROUP` is an alias for *CREATE ROLE*.

Compatibility

There is no `CREATE GROUP` statement in the SQL standard.

See Also

CREATE ROLE

CREATE INDEX

Defines a new index.

Synopsis

```
CREATE [UNIQUE] INDEX [name] ON table_name [USING method]
       ( {column_name | (expression)} [COLLATE parameter] [opclass] [ ASC |
 DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
       [ WITH ( storage_parameter = value [, ... ] ) ]
       [ TABLESPACE tablespace ]
       [ WHERE predicate ]
```

Description

`CREATE INDEX` constructs an index on the specified column(s) of the specified table or materialized view. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance).

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified if the index method supports multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on `upper(col)` would allow the clause `WHERE upper(col) = 'JIM'` to use an index.

Greenplum Database provides the index methods B-tree, bitmap, GiST, SP-GiST, and GIN. Users can also define their own index methods, but that is fairly complicated.

When the `WHERE` clause is present, a partial index is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet is most often selected, you can improve performance by creating an index on just that portion.

The expression used in the `WHERE` clause may refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Subqueries and aggregate expressions are also forbidden in `WHERE`. The same restrictions apply to index fields that are expressions.

All functions and operators used in an index definition must be immutable. Their results must depend only on their arguments and never on any outside influence (such as the contents of another table or a

parameter value). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression or `WHERE` clause, remember to mark the function `IMMUTABLE` when you create it.

Parameters

`UNIQUE`

Checks for duplicate values in the table when the index is created and each time data is added. Duplicate entries will generate an error. Unique indexes only apply to B-tree indexes. In Greenplum Database, unique indexes are allowed only if the columns of the index key are the same as (or a superset of) the Greenplum distribution key. On partitioned tables, a unique index is only supported within an individual partition - not across all partitions.

`name`

The name of the index to be created. The index is always created in the same schema as its parent table. If the name is omitted, Greenplum Database chooses a suitable name based on the parent table's name and the indexed column name(s).

`table_name`

The name (optionally schema-qualified) of the table to be indexed.

`method`

The name of the index method to be used. Choices are `btree`, `bitmap`, `gist`, `spgist`, and `gin`. The default method is `btree`.

Currently, only the B-tree, GiST, and GIN index methods support multicolumn indexes. Up to 32 fields can be specified by default. Only B-tree currently supports unique indexes.

GPORCA supports only B-tree, bitmap, GiST, and GIN indexes. GPORCA ignores indexes created with unsupported indexing methods.

`column_name`

The name of a column of the table on which to create the index. Only the B-tree, bitmap, GiST, and GIN index methods support multicolumn indexes.

`expression`

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses may be omitted if the expression has the form of a function call.

`collation`

The name of the collation to use for the index. By default, the index uses the collation declared for the column to be indexed or the result collation of the expression to be indexed. Indexes with non-default collations can be useful for queries that involve expressions using non-default collations.

`opclass`

The name of an operator class. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class (this operator class includes comparison functions for four-byte integers). In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, a complex-number data type could be sorted by either absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index.

`ASC`

Specifies ascending sort order (which is the default).

`DESC`

Specifies descending sort order.

NULLS FIRST

Specifies that nulls sort before non-nulls. This is the default when DESC is specified.

NULLS LAST

Specifies that nulls sort after non-nulls. This is the default when DESC is not specified.

storage_parameter

The name of an index-method-specific storage parameter. Each index method has its own set of allowed storage parameters.

FILLFACTOR - B-tree, bitmap, GiST, and SP-GiST index methods all accept this parameter. The FILLFACTOR for an index is a percentage that determines how full the index method will try to pack index pages. For B-trees, leaf pages are filled to this percentage during initial index build, and also when extending the index at the right (adding new largest key values). If pages subsequently become completely full, they will be split, leading to gradual degradation in the index's efficiency. B-trees use a default fillfactor of 90, but any integer value from 10 to 100 can be selected. If the table is static then fillfactor 100 is best to minimize the index's physical size, but for heavily updated tables a smaller fillfactor is better to minimize the need for page splits. The other index methods use fillfactor in different but roughly analogous ways; the default fillfactor varies between methods.

BUFFERING - In addition to FILLFACTOR, GiST indexes additionally accept the BUFFERING parameter. BUFFERING determines whether Greenplum Database builds the index using the buffering build technique described in *GiST buffering build* in the PostgreSQL documentation. With OFF it is disabled, with ON it is enabled, and with AUTO it is initially disabled, but turned on on-the-fly once the index size reaches *effective-cache-size*. The default is AUTO.

FASTUPDATE - The GIN index method accepts the FASTUPDATE storage parameter. FASTUPDATE is a Boolean parameter that disables or enables the GIN index fast update technique. A value of ON enables fast update (the default), and OFF disables it. See *GIN fast update technique* in the PostgreSQL documentation for more information.

> **Note:** Turning FASTUPDATE off via ALTER INDEX prevents future insertions from going into the list of pending index entries, but does not in itself flush previous entries. You might want to VACUUM the table afterward to ensure the pending list is emptied.

tablespace_name

The tablespace in which to create the index. If not specified, the default tablespace is used, or *temp_tablespaces* for indexes on temporary tables.

predicate

The constraint expression for a partial index.

Notes

An *operator class* can be specified for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the int4_ops class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index.

For index methods that support ordered scans (currently, only B-tree), the optional clauses ASC, DESC, NULLS FIRST, and/or NULLS LAST can be specified to modify the sort ordering of the index. Since an

ordered index can be scanned either forward or backward, it is not normally useful to create a single-column `DESC` index — that sort ordering is already available with a regular index. The value of these options is that multicolumn indexes can be created that match the sort ordering requested by a mixed-ordering query, such as `SELECT ... ORDER BY x ASC, y DESC`. The `NULLS` options are useful if you need to support "nulls sort low" behavior, rather than the default "nulls sort high", in queries that depend on indexes to avoid sorting steps.

For most index methods, the speed of creating an index is dependent on the setting of `maintenance_work_mem`. Larger values will reduce the time needed for index creation, so long as you don't make it larger than the amount of memory really available, which would drive the machine into swapping.

When an index is created on a partitioned table, the index is propagated to all the child tables created by Greenplum Database. Creating an index on a table that is created by Greenplum Database for use by a partitioned table is not supported.

`UNIQUE` indexes are allowed only if the index columns are the same as (or a superset of) the Greenplum distribution key columns.

`UNIQUE` indexes are not allowed on append-optimized tables.

A `UNIQUE` index can be created on a partitioned table. However, uniqueness is enforced only within a partition; uniqueness is not enforced between partitions. For example, for a partitioned table with partitions that are based on year and a subpartitions that are based on quarter, uniqueness is enforced only on each individual quarter partition. Uniqueness is not enforced between quarter partitions

Indexes are not used for `IS NULL` clauses by default. The best way to use indexes in such cases is to create a partial index using an `IS NULL` predicate.

`bitmap` indexes perform best for columns that have between 100 and 100,000 distinct values. For a column with more than 100,000 distinct values, the performance and space efficiency of a bitmap index decline. The size of a bitmap index is proportional to the number of rows in the table times the number of distinct values in the indexed column.

Columns with fewer than 100 distinct values usually do not benefit much from any type of index. For example, a gender column with only two distinct values for male and female would not be a good candidate for an index.

Prior releases of Greenplum Database also had an R-tree index method. This method has been removed because it had no significant advantages over the GiST method. If `USING rtree` is specified, `CREATE INDEX` will interpret it as `USING gist`.

For more information on the GiST index type, refer to the *PostgreSQL documentation*.

The use of hash indexes has been disabled in Greenplum Database.

Examples

To create a B-tree index on the column `title` in the table `films`:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

To create a bitmap index on the column `gender` in the table `employee`:

```
CREATE INDEX gender_bmp_idx ON employee USING bitmap
(gender);
```

To create an index on the expression `lower(title)`, allowing efficient case-insensitive searches:

```
CREATE INDEX ON films ((lower(title)));
```

(In this example we have chosen to omit the index name, so the system will choose a name, typically `films_lower_idx`.)

To create an index with non-default collation:

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

To create an index with non-default fill factor:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH
(fillfactor = 70);
```

To create a GIN index with fast updates disabled:

```
CREATE INDEX gin_idx ON documents_table USING gin (locations) WITH
 (fastupdate = off);
```

To create an index on the column `code` in the table `films` and have the index reside in the tablespace `indexspace`:

```
CREATE INDEX code_idx ON films(code) TABLESPACE indexspace;
```

To create a GiST index on a point attribute so that we can efficiently use box operators on the result of the conversion function:

```
CREATE INDEX pointloc ON points USING gist (box(location,location));
SELECT * FROM points WHERE box(location,location) && '(0,0),(1,1)'::box;
```

Compatibility

`CREATE INDEX` is a Greenplum Database language extension. There are no provisions for indexes in the SQL standard.

Greenplum Database does not support the concurrent creation of indexes (`CONCURRENTLY` keyword not supported).

See Also

`ALTER INDEX`, *`DROP INDEX`*, *`CREATE TABLE`*, *`CREATE OPERATOR CLASS`*

CREATE LANGUAGE

Defines a new procedural language.

Synopsis

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name

CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ]
   [ VALIDATOR valfunction ]
```

Description

`CREATE LANGUAGE` registers a new procedural language with a Greenplum database. Subsequently, functions and trigger procedures can be defined in this new language.

> **Note:** Procedural languages for Greenplum Database have been made into "extensions," and should therefore be installed with *CREATE EXTENSION*, not `CREATE LANGUAGE`. Using `CREATE LANGUAGE` directly should be restricted to extension installation scripts. If you have a "bare" language in your database, perhaps as a result of an upgrade, you can convert it to an extension using `CREATE EXTENSION` *`langname`* `FROM unpackaged`.

Superusers can register a new language with a Greenplum database. A database owner can also register within that database any language listed in the `pg_pltemplate` catalog in which the `tmpldbacreate` field is true. The default configuration allows only trusted languages to be registered by database owners. The creator of a language becomes its owner and can later drop it, rename it, or assign ownership to a new owner.

`CREATE OR REPLACE LANGUAGE` will either create a new language, or replace an existing definition. If the language already exists, its parameters are updated according to the values specified or taken from *pg_pltemplate*, but the language's ownership and permissions settings do not change, and any existing functions written in the language are assumed to still be valid. In addition to the normal privilege requirements for creating a language, the user must be superuser or owner of the existing language. The `REPLACE` case is mainly meant to be used to ensure that the language exists. If the language has a *pg_pltemplate* entry then `REPLACE` will not actually change anything about an existing definition, except in the unusual case where the *pg_pltemplate* entry has been modified since the language was created.

`CREATE LANGUAGE` effectively associates the language name with handler function(s) that are responsible for executing functions written in that language. For a function written in a procedural language (a language other than C or SQL), the database server has no built-in knowledge about how to interpret the function's source code. The task is passed to a special handler that knows the details of the language. The handler could either do all the work of parsing, syntax analysis, execution, and so on or it could serve as a bridge between Greenplum Database and an existing implementation of a programming language. The handler itself is a C language function compiled into a shared object and loaded on demand, just like any other C function. Therese procedural language packages are included in the standard Greenplum Database distribution: PL/pgSQL, PL/Perl, and PL/Python. Language handlers have also been added for PL/Java and PL/R, but those languages are not pre-installed with Greenplum Database. See the topic on *Procedural Languages* in the PostgreSQL documentation for more information on developing functions using these procedural languages.

The PL/Perl, PL/Java, and PL/R libraries require the correct versions of Perl, Java, and R to be installed, respectively.

On RHEL and SUSE platforms, download the appropriate extensions from *Pivotal Network*, then install the extensions using the Greenplum Package Manager (`gppkg`) utility to ensure that all dependencies are installed as well as the extensions. See the Greenplum Database Utility Guide for details about `gppkg`.

There are two forms of the `CREATE LANGUAGE` command. In the first form, the user specifies the name of the desired language and the Greenplum Database server uses the `pg_pltemplate` system catalog to determine the correct parameters. In the second form, the user specifies the language parameters as well as the language name. You can use the second form to create a language that is not defined in `pg_pltemplate`.

When the server finds an entry in the `pg_pltemplate` catalog for the given language name, it will use the catalog data even if the command includes language parameters. This behavior simplifies loading of old dump files, which are likely to contain out-of-date information about language support functions.

Parameters

TRUSTED

`TRUSTED` specifies that the language does not grant access to data that the user would not otherwise have. If this key word is omitted when registering the language, only users with the Greenplum Database superuser privilege can use this language to create new functions.

PROCEDURAL

This is a noise word.

name

The name of the new procedural language. The name must be unique among the languages in the database. Built-in support is included for `plpgsql`, `plperl`, and `plpythonu`. The languages `plpgsql` (PL/pgSQL) and `plpythonu` (PL/Python) are installed by default in Greenplum Database.

HANDLER *call_handler*

Ignored if the server has an entry for the specified language name in `pg_pltemplate`. The name of a previously registered function that will be called to execute the procedural language functions. The call handler for a procedural language must be written in a compiled language such as C with version 1 call convention and registered with Greenplum Database as a function taking no arguments and returning the `language_handler` type, a placeholder type that is simply used to identify the function as a call handler.

INLINE *inline_handler*

The name of a previously registered function that is called to execute an anonymous code block in this language that is created with the *DO* command. If an `inline_handler` function is not specified, the language does not support anonymous code blocks. The handler function must take one argument of type `internal`, which is the *DO* command internal representation. The function typically return `void`. The return value of the handler is ignored.

VALIDATOR *valfunction*

Ignored if the server has an entry for the specified language name in `pg_pltemplate`. The name of a previously registered function that will be called to execute the procedural language functions. The call handler for a procedural language must be written in a compiled language such as C with version 1 call convention and registered with Greenplum Database as a function taking no arguments and returning the `language_handler` type, a placeholder type that is simply used to identify the function as a call handler.

Notes

The PL/pgSQL language is already registered in all databases by default. The PL/Python language extension is installed but not registered.

The system catalog `pg_language` records information about the currently installed languages.

To create functions in a procedural language, a user must have the `USAGE` privilege for the language. By default, `USAGE` is granted to `PUBLIC` (everyone) for trusted languages. This may be revoked if desired.

Procedural languages are local to individual databases. You create and drop languages for individual databases.

The call handler function and the validator function (if any) must already exist if the server does not have an entry for the language in `pg_pltemplate`. But when there is an entry, the functions need not already exist; they will be automatically defined if not present in the database.

Any shared library that implements a language must be located in the same `LD_LIBRARY_PATH` location on all segment hosts in your Greenplum Database array.

Examples

The preferred way of creating any of the standard procedural languages is to use `CREATE EXTENSION` instead of `CREATE LANGUAGE`. For example:

```
CREATE EXTENSION plperl;
```

For a language not known in the `pg_pltemplate` catalog:

```
CREATE FUNCTION plsample_call_handler() RETURNS
language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Compatibility

`CREATE LANGUAGE` is a Greenplum Database extension.

See Also

ALTER LANGUAGE, *CREATE EXTENSION*, *CREATE FUNCTION*, *DROP EXTENSION*, *DROP LANGUAGE*, *GRANT DO*

CREATE MATERIALIZED VIEW

Defines a new materialized view.

Synopsis

```
CREATE MATERIALIZED VIEW table_name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
    [DISTRIBUTED {| BY column [opclass], [ ... ] | RANDOMLY | REPLICATED }]
```

Description

`CREATE MATERIALIZED VIEW` defines a materialized view of a query. The query is executed and used to populate the view at the time the command is issued (unless `WITH NO DATA` is used) and can be refreshed using `REFRESH MATERIALIZED VIEW`.

`CREATE MATERIALIZED VIEW` is similar to `CREATE TABLE AS`, except that it also remembers the query used to initialize the view, so that it can be refreshed later upon demand. To refresh materialized view data, use the `REFRESH MATERIALIZED VIEW` command. A materialized view has many of the same properties as a table, but there is no support for temporary materialized views or automatic generation of OIDs.

Parameters

table_name

The name (optionally schema-qualified) of the materialized view to be created.

column_name

The name of a column in the materialized view. The column names are assigned based on position. The first column name is assigned to the first column of the query result, and so on. If a column name is not provided, it is taken from the output column names of the query.

WITH (*storage_parameter* [= *value*] [, ...])

This clause specifies optional storage parameters for the materialized view. All parameters supported for `CREATE TABLE` are also supported for `CREATE MATERIALIZED VIEW` with the exception of OIDS. See *CREATE TABLE* for more information.

TABLESPACE *tablespace_name*

The *tablespace_name* is the name of the tablespace in which the new materialized view is to be created. If not specified, server configuration parameter *default_tablespace* is consulted.

query

A *SELECT* or *VALUES* command. This query will run within a security-restricted operation; in particular, calls to functions that themselves create temporary tables will fail.

WITH [NO] DATA

This clause specifies whether or not the materialized view should be populated with data at creation time. `WITH DATA` is the default, populate the materialized view. For `WITH NO DATA`, the materialized view is not populated with data, is flagged as unscannable, and cannot be queried until `REFRESH MATERIALIZED VIEW` is used to populate the materialized view. An error is returned if a query attempts to access an unscannable materialized view.

DISTRIBUTED BY (column [opclass], [...])
DISTRIBUTED RANDOMLY
DISTRIBUTED REPLICATED

Used to declare the Greenplum Database distribution policy for the materialized view data. For information about a table distribution policy, see *CREATE TABLE*.

Notes

Materialized views are read only. The system will not allow an `INSERT`, `UPDATE`, or `DELETE` on a materialized view. Use `REFRESH MATERIALIZED VIEW` to update the materialized view data.

If you want the data to be ordered upon generation, you must use an `ORDER BY` clause in the materialized view query. However, if a materialized view query contains an `ORDER BY` or `SORT` clause, the data is not guaranteed to be ordered or sorted if `SELECT` is performed on the materialized view.

Examples

Create a view consisting of all comedy films:

```
CREATE MATERIALIZED VIEW comedies AS SELECT * FROM films
WHERE kind = 'comedy';
```

This will create a view containing the columns that are in the `film` table at the time of view creation. Though * was used to create the materialized view, columns added later to the table will not be part of the view.

Create a view that gets the top ten ranked baby names:

```
CREATE MATERIALIZED VIEW topten AS SELECT name, rank, gender, year FROM
names, rank WHERE rank < '11' AND names.id=rank.id;
```

Compatibility

`CREATE MATERIALIZED VIEW` is a Greenplum Database extension of the SQL standard.

See Also

SELECT, *VALUES*, *CREATE VIEW*, *ALTER MATERIALIZED VIEW*, *DROP MATERIALIZED VIEW*, *REFRESH MATERIALIZED VIEW*

CREATE OPERATOR

Defines a new operator.

Synopsis

```
CREATE OPERATOR name (
       PROCEDURE = funcname
       [, LEFTARG = lefttype] [, RIGHTARG = righttype]
       [, COMMUTATOR = com_op] [, NEGATOR = neg_op]
       [, RESTRICT = res_proc] [, JOIN = join_proc]
       [, HASHES] [, MERGES] )
```

Description

`CREATE OPERATOR` defines a new operator. The user who defines an operator becomes its owner.

The operator name is a sequence of up to `NAMEDATALEN`-1 (63 by default) characters from the following list: `+ - * / < > = ~ ! @ # % ^ & | ` ?`

There are a few restrictions on your choice of name:

- `--` and `/*` cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multicharacter operator name cannot end in + or `-`, unless the name also contains at least one of these characters: `~ ! @ # % ^ & | ` ?`

For example, `@-` is an allowed operator name, but `*-` is not. This restriction allows Greenplum Database to parse SQL-compliant commands without requiring spaces between tokens.

The use of => as an operator name is deprecated. It may be disallowed altogether in a future release.

The operator `!=` is mapped to <> on input, so these two names are always equivalent.

At least one of `LEFTARG` and `RIGHTARG` must be defined. For binary operators, both must be defined. For right unary operators, only `LEFTARG` should be defined, while for left unary operators only `RIGHTARG` should be defined.

The *funcname* procedure must have been previously defined using `CREATE FUNCTION`, must be `IMMUTABLE`, and must be defined to accept the correct number of arguments (either one or two) of the indicated types.

The other clauses specify optional operator optimization clauses. These clauses should be provided whenever appropriate to speed up queries that use the operator. But if you provide them, you must be sure that they are correct. Incorrect use of an optimization clause can result in server process crashes, subtly wrong output, or other unexpected results. You can always leave out an optimization clause if you are not sure about it.

To be able to create an operator, you must have `USAGE` privilege on the argument types and the return type, as well as `EXECUTE` privilege on the underlying function. If a commutator or negator operator is specified, you must own these operators.

Parameters

name

The (optionally schema-qualified) name of the operator to be defined. Two operators in the same schema can have the same name if they operate on different data types.

funcname

The function used to implement this operator (must be an `IMMUTABLE` function).

lefttype

The data type of the operator's left operand, if any. This option would be omitted for a left-unary operator.

righttype

The data type of the operator's right operand, if any. This option would be omitted for a right-unary operator.

com_op

The optional `COMMUTATOR` clause names an operator that is the commutator of the operator being defined. We say that operator A is the commutator of operator B if (x A y) equals (y B x) for all possible input values x, y. Notice that B is also the commutator of A. For example, operators < and > for a particular data type are usually each others commutators, and operator + is usually commutative with itself. But operator - is usually not commutative with anything. The left operand type of a commutable operator is the same as the right operand type of its commutator, and vice versa. So the name of the commutator operator is all that needs to be provided in the `COMMUTATOR` clause.

neg_op

The optional `NEGATOR` clause names an operator that is the negator of the operator being defined. We say that operator A is the negator of operator B if both return Boolean results and (x A y) equals NOT (x B y) for all possible inputs x, y. Notice that B is also the negator of A. For example, < and >= are a negator pair for most data types. An operator's negator must have the same left and/or right operand types as the operator to be defined, so only the operator name need be given in the `NEGATOR` clause.

res_proc

The optional `RESTRICT` names a restriction selectivity estimation function for the operator. Note that this is a function name, not an operator name. `RESTRICT` clauses only make sense for binary operators that return `boolean`. The idea behind a restriction selectivity estimator is to guess what fraction of the rows in a table will satisfy a `WHERE`-clause condition of the form:

```
column OP constant
```

for the current operator and a particular constant value. This assists the optimizer by giving it some idea of how many rows will be eliminated by `WHERE` clauses that have this form.

You can usually just use one of the following system standard estimator functions for many of your own operators:

`eqsel` for =

`neqsel` for <>

`scalarltsel` for < or <=

`scalargtsel` for > or >=

join_proc

The optional `JOIN` clause names a join selectivity estimation function for the operator. Note that this is a function name, not an operator name. `JOIN` clauses only make sense for binary operators that return `boolean`. The idea behind a join selectivity estimator is to guess what fraction of the rows in a pair of tables will satisfy a `WHERE`-clause condition of the form

```
table1.column1 OP table2.column2
```

for the current operator. This helps the optimizer by letting it figure out which of several possible join sequences is likely to take the least work.

You can usually just use one of the following system standard join selectivity estimator functions for many of your own operators:

`eqjoinsel` for =

`neqjoinsel` for <>

`scalarltjoinsel` for < or <=

`scalargtjoinsel` for > or >=

`areajoinsel` for 2D area-based comparisons

`positionjoinsel` for 2D position-based comparisons

`contjoinsel` for 2D containment-based comparisons

HASHES

The optional HASHES clause tells the system that it is permissible to use the hash join method for a join based on this operator. HASHES only makes sense for a binary operator that returns `boolean`. The hash join operator can only return true for pairs of left and right values that hash to the same hash code. If two values are put in different hash buckets, the join will never compare them, implicitly assuming that the result of the join operator must be false. Because of this, it never makes sense to specify HASHES for operators that do not represent equality.

In most cases, it is only practical to support hashing for operators that take the same data type on both sides. However, you can design compatible hash functions for two or more data types, which are functions that will generate the same hash codes for "equal" values, even if the values are differently represented.

To be marked HASHES, the join operator must appear in a hash index operator class. Attempts to use the operator in hash joins will fail at run time if no such operator class exists. The system needs the operator class to find the data-type-specific hash function for the operator's input data type. You must also supply a suitable hash function before you can create the operator class. Exercise care when preparing a hash function, as there are machine-dependent ways in which it could fail to function correctly. For example, on machines that meet the IEEE floating-point standard, negative zero and positive zero are different values (different bit patterns) but are defined to compare as equal. If a float value could contain a negative zero, define it to generate the same hash value as positive zero.

A hash-joinable operator must have a commutator (itself, if the two operand data types are the same, or a related equality operator if they are different) that appears in the same operator family. Otherwise, planner errors can occur when the operator is used. For better optimization, a hash operator family that supports multiple data types should provide equality operators for every combination of the data types.

> **Note:** The function underlying a hash-joinable operator must be marked immutable or stable; an operator marked as volatile will not be used. If a hash-joinable operator has an underlying function that is marked strict, the function must also be complete, returning true or false, and not null, for any two non-null inputs.

MERGES

The MERGES clause, if present, tells the system that it is permissible to use the merge-join method for a join based on this operator. MERGES only makes sense for a binary operator that returns `boolean`, and in practice the operator must represent equality for some data type or pair of data types.

Merge join is based on the idea of sorting the left- and right-hand tables into order and then scanning them in parallel. This means both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at equivalent places in the sort order. In practice, this means that the join operator must

behave like an equality operator. However, you can merge-join two distinct data types so long as they are logically compatible. For example, the `smallint-versus-integer` equality operator is merge-joinable. Only sorting operators that bring both data types into a logically compatible sequence are needed.

To be marked `MERGES`, the join operator must appear as an equality member of a btree index operator family. This is not enforced when you create the operator, because the referencing operator family does not exist until later. However, the operator will not actually be used for merge joins unless a matching operator family can be found. The `MERGE` flag thus acts as a suggestion to the planner to look for a matching operator family.

A merge-joinable operator must have a commutator that appears in the same operator family. This would be itself, if the two operand data types are the same, or a related equality operator if the data types are different. Without an appropriate commutator, planner errors can occur when the operator is used. Also, although not strictly required, a btree operator family that supports multiple data types should be able to provide equality operators for every combination of the data types; this allows better optimization.

> **Note:** `SORT1`, `SORT2`, `LTCMP`, and `GTCMP` were formerly used to specify the names of sort operators associated with a merge-joinable operator. Information about associated operators is now found by looking at B-tree operator families; specifying any of these operators will be ignored, except that it will implicitly set `MERGES` to true.

Notes

Any functions used to implement the operator must be defined as `IMMUTABLE`.

It is not possible to specify an operator's lexical precedence in `CREATE OPERATOR`, because the parser's precedence behavior is hard-wired. See *Operator Precedence* in the PostgreSQL documentation for precedence details.

Use *`DROP OPERATOR`* to delete user-defined operators from a database. Use *`ALTER OPERATOR`* to modify operators in a database.

Examples

Here is an example of creating an operator for adding two complex numbers, assuming we have already created the definition of type `complex`. First define the function that does the work, then define the operator:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;
CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

To use this operator in a query:

```
SELECT (a + b) AS c FROM test_complex;
```

Compatibility

`CREATE OPERATOR` is a Greenplum Database language extension. The SQL standard does not provide for user-defined operators.

See Also

CREATE FUNCTION, *CREATE TYPE*, *ALTER OPERATOR*, *DROP OPERATOR*

CREATE OPERATOR CLASS

Defines a new operator class.

Synopsis

```
CREATE OPERATOR CLASS name [DEFAULT] FOR TYPE data_type
  USING index_method [ FAMILY family_name ] AS
  { OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ FOR
 SEARCH | FOR ORDER BY sort_family_name ]
  | FUNCTION support_number funcname (argument_type [, ...] )
  | STORAGE storage_type
  } [, ... ]
```

Description

`CREATE OPERATOR CLASS` creates a new operator class. An operator class defines how a particular data type can be used with an index. The operator class specifies that certain operators will fill particular roles or strategies for this data type and this index method. The operator class also specifies the support procedures to be used by the index method when the operator class is selected for an index column. All the operators and functions used by an operator class must be defined before the operator class is created. Any functions used to implement the operator class must be defined as `IMMUTABLE`.

`CREATE OPERATOR CLASS` does not presently check whether the operator class definition includes all the operators and functions required by the index method, nor whether the operators and functions form a self-consistent set. It is the user's responsibility to define a valid operator class.

You must be a superuser to create an operator class.

Parameters

name

The (optionally schema-qualified) name of the operator class to be defined. Two operator classes in the same schema can have the same name only if they are for different index methods.

DEFAULT

Makes the operator class the default operator class for its data type. At most one operator class can be the default for a specific data type and index method.

data_type

The column data type that this operator class is for.

index_method

The name of the index method this operator class is for. Choices are `btree`, `bitmap`, and `gist`.

family_name

The name of the existing operator family to add this operator class to. If not specified, a family named the same as the operator class is used (creating it, if it doesn't already exist).

strategy_number

The operators associated with an operator class are identified by *strategy numbers*, which serve to identify the semantics of each operator within the context of its operator class. For example, B-trees impose a strict ordering on keys, lesser to greater, and so operators like *less than* and *greater than or equal to* are interesting with respect to a B-tree. These

strategies can be thought of as generalized operators. Each operator class specifies which actual operator corresponds to each strategy for a particular data type and interpretation of the index semantics. The corresponding strategy numbers for each index method are as follows:

Table 83: B-tree and Bitmap Strategies

Operation	Strategy Number
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

Table 84: GiST Two-Dimensional Strategies (R-Tree)

Operation	Strategy Number
strictly left of	1
does not extend to right of	2
overlaps	3
does not extend to left of	4
strictly right of	5
same	6
contains	7
contained by	8
does not extend above	9
strictly below	10
strictly above	11
does not extend below	12

sort_family_name

The name (optionally schema-qualified) of an existing `btree` operator family that describes the sort ordering associated with an ordering operator.

If neither `FOR SEARCH` nor `FOR ORDER BY` is specified, `FOR SEARCH` is the default.

operator_name

The name (optionally schema-qualified) of an operator associated with the operator class.

op_type

In an `OPERATOR` clause, the operand data type(s) of the operator, or `NONE` to signify a left-unary or right-unary operator. The operand data types can be omitted in the normal case where they are the same as the operator class's data type.

In a `FUNCTION` clause, the operand data type(s) the function is intended to support, if different from the input data type(s) of the function (for B-tree comparison functions and hash functions) or the class's data type (for B-tree sort support functions and all functions

in GiST, SP-GiST, and GIN operator classes). These defaults are correct, and so *op_type* need not be specified in `FUNCTION` clauses, except for the case of a B-tree sort support function that is meant to support cross-data-type comparisons.

support_number

Index methods require additional support routines in order to work. These operations are administrative routines used internally by the index methods. As with strategies, the operator class identifies which specific functions should play each of these roles for a given data type and semantic interpretation. The index method defines the set of functions it needs, and the operator class identifies the correct functions to use by assigning them to the *support function numbers* as follows:

Table 85: B-tree and Bitmap Support Functions

Function	Support Number
Compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second.	1

Table 86: GiST Support Functions

Function	Support Number
consistent - determine whether key satisfies the query qualifier.	1
union - compute union of a set of keys.	2
compress - compute a compressed representation of a key or value to be indexed.	3
decompress - compute a decompressed representation of a compressed key.	4
penalty - compute penalty for inserting new key into subtree with given subtree's key.	5
picksplit - determine which entries of a page are to be moved to the new page and compute the union keys for resulting pages.	6
equal - compare two keys and return true if they are equal.	7

funcname

The name (optionally schema-qualified) of a function that is an index method support procedure for the operator class.

argument_types

The parameter data type(s) of the function.

storage_type

The data type actually stored in the index. Normally this is the same as the column data type, but some index methods (currently GiST and GIN) allow it to be different. The `STORAGE` clause must be omitted unless the index method allows a different type to be used.

Notes

Because the index machinery does not check access permissions on functions before using them, including a function or operator in an operator class is the same as granting public execute permission on it. This is usually not an issue for the sorts of functions that are useful in an operator class.

The operators should not be defined by SQL functions. A SQL function is likely to be inlined into the calling query, which will prevent the optimizer from recognizing that the query matches an index.

Any functions used to implement the operator class must be defined as `IMMUTABLE`.

Before Greenplum Database 6.0, the `OPERATOR` clause could include a `RECHECK` option. This option is no longer supported. Greenplum Database now determines whether an index operator is "lossy" on-the-fly at run time. This allows more efficient handling of cases where an operator might or might not be lossy.

Examples

The following example command defines a GiST index operator class for the data type `_int4` (array of int4). See the `intarray` contrib module for the complete example.

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR 3 &&,
        OPERATOR 6 = (anyarray, anyarray),
        OPERATOR 7 @>,
        OPERATOR 8 <@,
        OPERATOR 20 @@ (_int4, query_int),
        FUNCTION 1 g_int_consistent (internal, _int4, int, oid, internal),
        FUNCTION 2 g_int_union (internal, internal),
        FUNCTION 3 g_int_compress (internal),
        FUNCTION 4 g_int_decompress (internal),
        FUNCTION 5 g_int_penalty (internal, internal, internal),
        FUNCTION 6 g_int_picksplit (internal, internal),
        FUNCTION 7 g_int_same (_int4, _int4, internal);
```

Compatibility

`CREATE OPERATOR CLASS` is a Greenplum Database extension. There is no `CREATE OPERATOR CLASS` statement in the SQL standard.

See Also

`ALTER OPERATOR CLASS`, *`DROP OPERATOR CLASS`*, *`CREATE FUNCTION`*

CREATE OPERATOR FAMILY

Defines a new operator family.

Synopsis

```
CREATE OPERATOR FAMILY name  USING index_method
```

Description

`CREATE OPERATOR FAMILY` creates a new operator family. An operator family defines a collection of related operator classes, and perhaps some additional operators and support functions that are compatible with these operator classes but not essential for the functioning of any individual index. (Operators and functions that are essential to indexes should be grouped within the relevant operator class, rather than being "loose" in the operator family. Typically, single-data-type operators are bound to operator classes,

while cross-data-type operators can be loose in an operator family containing operator classes for both data types.)

The new operator family is initially empty. It should be populated by issuing subsequent `CREATE OPERATOR CLASS` commands to add contained operator classes, and optionally `ALTER OPERATOR FAMILY`commands to add "loose" operators and their corresponding support functions.

If a schema name is given then the operator family is created in the specified schema. Otherwise it is created in the current schema. Two operator families in the same schema can have the same name only if they are for different index methods.

The user who defines an operator family becomes its owner. Presently, the creating user must be a superuser. (This restriction is made because an erroneous operator family definition could confuse or even crash the server.)

Parameters

name

The (optionally schema-qualified) name of the operator family to be defined. The name can be schema-qualified.

index_method

The name of the index method this operator family is for.

Compatibility

`CREATE OPERATOR FAMILY` is a Greenplum Database extension. There is no `CREATE OPERATOR FAMILY` statement in the SQL standard.

See Also

`ALTER OPERATOR FAMILY`, *`DROP OPERATOR FAMILY`*, *`CREATE FUNCTION`*, *`ALTER OPERATOR CLASS`*, *`CREATE OPERATOR CLASS`*, *`DROP OPERATOR CLASS`*

CREATE PROTOCOL

Registers a custom data access protocol that can be specified when defining a Greenplum Database external table.

Synopsis

```
CREATE [TRUSTED] PROTOCOL name (
   [readfunc='read_call_handler'] [, writefunc='write_call_handler']
   [, validatorfunc='validate_handler' ])
```

Description

`CREATE PROTOCOL` associates a data access protocol name with call handlers that are responsible for reading from and writing data to an external data source. You must be a superuser to create a protocol.

The `CREATE PROTOCOL` command must specify either a read call handler or a write call handler. The call handlers specified in the `CREATE PROTOCOL` command must be defined in the database.

The protocol name can be specified in a `CREATE EXTERNAL TABLE` command.

For information about creating and enabling a custom data access protocol, see "Example Custom Data Access Protocol" in the *Greenplum Database Administrator Guide*.

Parameters

TRUSTED

If not specified, only superusers and the protocol owner can create external tables using the protocol. If specified, superusers and the protocol owner can `GRANT` permissions on the protocol to other database roles.

name

The name of the data access protocol. The protocol name is case sensitive. The name must be unique among the protocols in the database.

readfunc= '*read_call_handler*'

The name of a previously registered function that Greenplum Database calls to read data from an external data source. The command must specify either a read call handler or a write call handler.

writefunc= '*write_call_handler*'

The name of a previously registered function that Greenplum Database calls to write data to an external data source. The command must specify either a read call handler or a write call handler.

validatorfunc='*validate_handler*'

An optional validator function that validates the URL specified in the `CREATE EXTERNAL TABLE` command.

Notes

Greenplum Database handles external tables of type `file`, `gpfdist`, and `gpfdists` internally. See *Configuring and Using S3 External Tables* for information about enabling the S3 protocol. Refer to *pxf:// Protocol* for information about using the `pxf` protocol.

Any shared library that implements a data access protocol must be located in the same location on all Greenplum Database segment hosts. For example, the shared library can be in a location specified by the operating system environment variable `LD_LIBRARY_PATH` on all hosts. You can also specify the location when you define the handler function. For example, when you define the `s3` protocol in the `CREATE PROTOCOL` command, you specify `$libdir/gps3ext.so` as the location of the shared object, where `$libdir` is located at `$GPHOME/lib`.

Compatibility

`CREATE PROTOCOL` is a Greenplum Database extension.

See Also

`ALTER PROTOCOL`, *`CREATE EXTERNAL TABLE`*, *`DROP PROTOCOL`*, *`GRANT`*

CREATE RESOURCE GROUP

Defines a new resource group.

Synopsis

```
CREATE RESOURCE GROUP name WITH (group_attribute=value [, ... ])
```

where *group_attribute* is:

```
CPU_RATE_LIMIT=integer | CPUSET=tuple
[ MEMORY_LIMIT=integer ]
[ CONCURRENCY=integer ]
[ MEMORY_SHARED_QUOTA=integer ]
[ MEMORY_SPILL_RATIO=integer ]
[ MEMORY_AUDITOR= {vmtracker | cgroup} ]
```

Description

Creates a new resource group for Greenplum Database resource management. You can create resource groups to manage resources for roles or to manage the resources of a Greenplum Database external component such as PL/Container.

A resource group that you create to manage a user role identifies concurrent transaction, memory, and CPU limits for the role when resource groups are enabled. You may assign such resource groups to one or more roles.

A resource group that you create to manage the resources of a Greenplum Database external component such as PL/Container identifies the memory and CPU limits for the component when resource groups are enabled. These resource groups use cgroups for both CPU and memory management. Assignment of resource groups to external components is component-specific. For example, you assign a PL/Container resource group when you configure a PL/Container runtime. You cannot assign a resource group that you create for external components to a role, nor can you assign a resource group that you create for roles to an external component.

You must have SUPERUSER privileges to create a resource group. The maximum number of resource groups allowed in your Greenplum Database cluster is 100.

Greenplum Database pre-defines two default resource groups: `admin_group` and `default_group`. These group names, as well as the group name `none`, are reserved.

To set appropriate limits for resource groups, the Greenplum Database administrator must be familiar with the queries typically executed on the system, as well as the users/roles executing those queries and the external components they may be using, such as PL/Containers.

After creating a resource group for a role, assign the group to one or more roles using the *`ALTER ROLE`* or *`CREATE ROLE`* commands.

After you create a resource group to manage the CPU and memory resources of an external component, configure the external component to use the resource group. For example, configure the PL/Container runtime `resource_group_id`.

Parameters

`name`

The name of the resource group.

`CONCURRENCY` ***`integer`***

The maximum number of concurrent transactions, including active and idle transactions, that are permitted for this resource group. The `CONCURRENCY` value must be an integer in the range [0 .. `max_connections`]. The default `CONCURRENCY` value for resource groups defined for roles is 20.

You must set `CONCURRENCY` to zero (0) for resource groups that you create for external components.

Note: You cannot set the `CONCURRENCY` value for the `admin_group` to zero (0).

`CPU_RATE_LIMIT` ***`integer`***
`CPUSET` ***`tuple`***

Required. You must specify only one of `CPU_RATE_LIMIT` or `CPUSET` when you create a resource group.

`CPU_RATE_LIMIT` is the percentage of CPU resources to allocate to this resource group. The minimum CPU percentage you can specify for a resource group is 1. The maximum is 100. The sum of the `CPU_RATE_LIMIT` values specified for all resource groups defined in the Greenplum Database cluster must be less than or equal to 100.

CPUSET identifies the CPU cores to reserve for this resource group. The CPU cores that you specify in *tuple* must be available in the system and cannot overlap with any CPU cores that you specify for other resource groups.

tuple is a comma-separated list of single core numbers or core number intervals. You must enclose *tuple* in single quotes, for example, '1,3-4'.

> **Note:** You can configure CPUSET for a resource group only after you have enabled resource group-based resource management for your Greenplum Database cluster.

MEMORY_LIMIT *integer*

The total percentage of Greenplum Database memory resources to reserve for this resource group. The minimum memory percentage you can specify for a resource group is 0. The maximum is 100. The default value is 0.

When you specify a MEMORY_LIMIT of 0, Greenplum Database reserves no memory for the resource group, but uses global shared memory to fulfill all memory requests in the group. If MEMORY_LIMIT is 0, MEMORY_SPILL_RATIO must also be 0.

The sum of the MEMORY_LIMIT values specified for all resource groups defined in the Greenplum Database cluster must be less than or equal to 100.

MEMORY_SHARED_QUOTA *integer*

The quota of shared memory in the resource group. Resource groups with a MEMORY_SHARED_QUOTA threshold set aside a percentage of memory allotted to the resource group to share across transactions. This shared memory is allocated on a first-come, first-served basis as available. A transaction may use none, some, or all of this memory. The minimum memory shared quota percentage you can specify for a resource group is 0. The maximum is 100. The default MEMORY_SHARED_QUOTA value is 80.

MEMORY_SPILL_RATIO *integer*

The memory usage threshold for memory-intensive operators in a transaction. When this threshold is reached, a transaction spills to disk. You can specify an integer percentage value from 0 to 100 inclusive. The default MEMORY_SPILL_RATIO value is 0. When MEMORY_SPILL_RATIO is 0, Greenplum Database uses the *statement_mem* server configuration parameter value to control initial query operator memory.

MEMORY_AUDITOR {vmtracker | cgroup}

The memory auditor for the resource group. Greenplum Database employs virtual memory tracking for role resources and cgroup memory tracking for resources used by external components. The default MEMORY_AUDITOR is vmtracker. When you create a resource group with vmtracker memory auditing, Greenplum Database tracks that resource group's memory internally.

When you create a resource group specifying the cgroup MEMORY_AUDITOR, Greenplum Database defers the accounting of memory used by that resource group to cgroups. CONCURRENCY must be zero (0) for a resource group that you create for external components such as PL/Container. You cannot assign a resource group that you create for external components to a Greenplum Database role.

Notes

You cannot submit a CREATE RESOURCE GROUP command in an explicit transaction or sub-transaction.

Use the gp_toolkit.gp_resgroup_config system view to display the limit settings of all resource groups:

```
SELECT * FROM gp_toolkit.gp_resgroup_config;
```

Examples

Create a resource group with CPU and memory limit percentages of 35:

```
CREATE RESOURCE GROUP rgroup1 WITH (CPU_RATE_LIMIT=35, MEMORY_LIMIT=35);
```

Create a resource group with a concurrent transaction limit of 20, a memory limit of 15, and a CPU limit of 25:

```
CREATE RESOURCE GROUP rgroup2 WITH (CONCURRENCY=20,
  MEMORY_LIMIT=15, CPU_RATE_LIMIT=25);
```

Create a resource group to manage PL/Container resources specifying a memory limit of 10, and a CPU limit of 10:

```
CREATE RESOURCE GROUP plc_run1 WITH (MEMORY_LIMIT=10, CPU_RATE_LIMIT=10,
  CONCURRENCY=0, MEMORY_AUDITOR=cgroup);
```

Create a resource group with a memory limit percentage of 11 to which you assign CPU cores 1 to 3:

```
CREATE RESOURCE GROUP rgroup3 WITH (CPUSET='1-3', MEMORY_LIMIT=11);
```

Compatibility

`CREATE RESOURCE GROUP` is a Greenplum Database extension. There is no provision for resource groups or resource management in the SQL standard.

See Also

`ALTER ROLE`, *`CREATE ROLE`*, *`ALTER RESOURCE GROUP`*, *`DROP RESOURCE GROUP`*

CREATE RESOURCE QUEUE

Defines a new resource queue.

Synopsis

```
CREATE RESOURCE QUEUE name WITH (queue_attribute=value [, ... ])
```

where *queue_attribute* is:

```
    ACTIVE_STATEMENTS=integer
        [ MAX_COST=float [COST_OVERCOMMIT={TRUE|FALSE}] ]
        [ MIN_COST=float ]
        [ PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX} ]
        [ MEMORY_LIMIT='memory_units' ]

  | MAX_COST=float [ COST_OVERCOMMIT={TRUE|FALSE} ]
        [ ACTIVE_STATEMENTS=integer ]
        [ MIN_COST=float ]
        [ PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX} ]
        [ MEMORY_LIMIT='memory_units' ]
```

Description

Creates a new resource queue for Greenplum Database resource management. A resource queue must have either an `ACTIVE_STATEMENTS` or a `MAX_COST` value (or it can have both). Only a superuser can create a resource queue.

Resource queues with an ACTIVE_STATEMENTS threshold set a maximum limit on the number of queries that can be executed by roles assigned to that queue. It controls the number of active queries that are allowed to run at the same time. The value for ACTIVE_STATEMENTS should be an integer greater than 0.

Resource queues with a MAX_COST threshold set a maximum limit on the total cost of queries that can be executed by roles assigned to that queue. Cost is measured in the *estimated total cost* for the query as determined by the query planner (as shown in the EXPLAIN output for a query). Therefore, an administrator must be familiar with the queries typically executed on the system in order to set an appropriate cost threshold for a queue. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for MAX_COST is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2). If a resource queue is limited based on a cost threshold, then the administrator can allow COST_OVERCOMMIT=TRUE (the default). This means that a query that exceeds the allowed cost threshold will be allowed to run but only when the system is idle. If COST_OVERCOMMIT=FALSE is specified, queries that exceed the cost limit will always be rejected and never allowed to run. Specifying a value for MIN_COST allows the administrator to define a cost for small queries that will be exempt from resource queueing.

> **Note:** GPORCA and the Postgres Planner utilize different query costing models and may compute different costs for the same query. The Greenplum Database resource queue resource management scheme neither differentiates nor aligns costs between GPORCA and the Postgres Planner; it uses the literal cost value returned from the optimizer to throttle queries.
>
> When resource queue-based resource management is active, use the MEMORY_LIMIT and ACTIVE_STATEMENTS limits for resource queues rather than configuring cost-based limits. Even when using GPORCA, Greenplum Database may fall back to using the Postgres Planner for certain queries, so using cost-based limits can lead to unexpected results.

If a value is not defined for ACTIVE_STATEMENTS or MAX_COST, it is set to -1 by default (meaning no limit). After defining a resource queue, you must assign roles to the queue using the *ALTER ROLE* or *CREATE ROLE* command.

You can optionally assign a PRIORITY to a resource queue to control the relative share of available CPU resources used by queries associated with the queue in relation to other resource queues. If a value is not defined for PRIORITY, queries associated with the queue have a default priority of MEDIUM.

Resource queues with an optional MEMORY_LIMIT threshold set a maximum limit on the amount of memory that all queries submitted through a resource queue can consume on a segment host. This determines the total amount of memory that all worker processes of a query can consume on a segment host during query execution. Greenplum recommends that MEMORY_LIMIT be used in conjunction with ACTIVE_STATEMENTS rather than with MAX_COST. The default amount of memory allotted per query on statement-based queues is: MEMORY_LIMIT / ACTIVE_STATEMENTS. The default amount of memory allotted per query on cost-based queues is: MEMORY_LIMIT * (query_cost / MAX_COST).

The default memory allotment can be overridden on a per-query basis using the statement_mem server configuration parameter, provided that MEMORY_LIMIT or max_statement_mem is not exceeded. For example, to allocate more memory to a particular query:

```
=> SET statement_mem='2GB';
=> SELECT * FROM my_big_table WHERE column='value' ORDER BY id;
=> RESET statement_mem;
```

The MEMORY_LIMIT value for all of your resource queues should not exceed the amount of physical memory of a segment host. If workloads are staggered over multiple queues, memory allocations can be oversubscribed. However, queries can be cancelled during execution if the segment host memory limit specified in gp_vmem_protect_limit is exceeded.

For information about statement_mem, max_statement, and gp_vmem_protect_limit, see *Server Configuration Parameters*.

Parameters

name

The name of the resource queue.

ACTIVE_STATEMENTS *integer*

Resource queues with an ACTIVE_STATEMENTS threshold limit the number of queries that can be executed by roles assigned to that queue. It controls the number of active queries that are allowed to run at the same time. The value for ACTIVE_STATEMENTS should be an integer greater than 0.

MEMORY_LIMIT *'memory_units'*

Sets the total memory quota for all statements submitted from users in this resource queue. Memory units can be specified in kB, MB or GB. The minimum memory quota for a resource queue is 10MB. There is no maximum, however the upper boundary at query execution time is limited by the physical memory of a segment host. The default is no limit (-1).

MAX_COST *float*

Resource queues with a MAX_COST threshold set a maximum limit on the total cost of queries that can be executed by roles assigned to that queue. Cost is measured in the *estimated total cost* for the query as determined by the Greenplum Database query optimizer (as shown in the EXPLAIN output for a query). Therefore, an administrator must be familiar with the queries typically executed on the system in order to set an appropriate cost threshold for a queue. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for MAX_COST is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2).

COST_OVERCOMMIT *boolean*

If a resource queue is limited based on MAX_COST, then the administrator can allow COST_OVERCOMMIT (the default). This means that a query that exceeds the allowed cost threshold will be allowed to run but only when the system is idle. If COST_OVERCOMMIT=FALSE is specified, queries that exceed the cost limit will always be rejected and never allowed to run.

MIN_COST *float*

The minimum query cost limit of what is considered a small query. Queries with a cost under this limit will not be queued and run immediately. Cost is measured in the *estimated total cost* for the query as determined by the query planner (as shown in the EXPLAIN output for a query). Therefore, an administrator must be familiar with the queries typically executed on the system in order to set an appropriate cost for what is considered a small query. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for MIN_COST is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2).

PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX}

Sets the priority of queries associated with a resource queue. Queries or statements in queues with higher priority levels will receive a larger share of available CPU resources in case of contention. Queries in low-priority queues may be delayed while higher priority queries are executed. If no priority is specified, queries associated with the queue have a priority of MEDIUM.

Notes

Use the gp_toolkit.gp_resqueue_status system view to see the limit settings and current status of a resource queue:

```
SELECT * from gp_toolkit.gp_resqueue_status WHERE
  rsqname='queue_name';
```

There is also another system view named `pg_stat_resqueues` which shows statistical metrics for a resource queue over time. To use this view, however, you must enable the `stats_queue_level` server configuration parameter. See "Managing Workload and Resources" in the *Greenplum Database Administrator Guide* for more information about using resource queues.

`CREATE RESOURCE QUEUE` cannot be run within a transaction.

Also, an SQL statement that is run during the execution of an `EXPLAIN ANALYZE` command is excluded from resource queues.

Examples

Create a resource queue with an active query limit of 20:

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20);
```

Create a resource queue with an active query limit of 20 and a total memory limit of 2000MB (each query will be allocated 100MB of segment host memory at execution time):

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20,
  MEMORY_LIMIT='2000MB');
```

Create a resource queue with a query cost limit of 3000.0:

```
CREATE RESOURCE QUEUE myqueue WITH (MAX_COST=3000.0);
```

Create a resource queue with a query cost limit of 3^{10} (or 30000000000.0) and do not allow overcommit. Allow small queries with a cost under 500 to run immediately:

```
CREATE RESOURCE QUEUE myqueue WITH (MAX_COST=3e+10,
  COST_OVERCOMMIT=FALSE, MIN_COST=500.0);
```

Create a resource queue with both an active query limit and a query cost limit:

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=30,
  MAX_COST=5000.00);
```

Create a resource queue with an active query limit of 5 and a maximum priority setting:

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=5,
  PRIORITY=MAX);
```

Compatibility

`CREATE RESOURCE QUEUE` is a Greenplum Database extension. There is no provision for resource queues or resource management in the SQL standard.

See Also

`ALTER ROLE`, *`CREATE ROLE`*, *`ALTER RESOURCE QUEUE`*, *`DROP RESOURCE QUEUE`*

CREATE ROLE

Defines a new database role (user or group).

Synopsis

```
CREATE ROLE name [[WITH] option [ ... ]]
```

where *option* can be:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | CREATEEXTTABLE | NOCREATEEXTTABLE
      [ ( attribute='value'[, ...] ) ]
           where attributes and value are:
           type='readable'|'writable'
           protocol='gpfdist'|'http'
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | ROLE rolename [, ...]
    | ADMIN rolename [, ...]
    | USER rolename [, ...]
    | SYSID uid [, ...]
    | RESOURCE QUEUE queue_name
    | RESOURCE GROUP group_name
    | [ DENY deny_point ]
    | [ DENY BETWEEN deny_point AND deny_point]
```

Description

CREATE ROLE adds a new role to a Greenplum Database system. A role is an entity that can own database objects and have database privileges. A role can be considered a user, a group, or both depending on how it is used. You must have CREATEROLE privilege or be a database superuser to use this command.

Note that roles are defined at the system-level and are valid for all databases in your Greenplum Database system.

Parameters

name

The name of the new role.

SUPERUSER
NOSUPERUSER

If SUPERUSER is specified, the role being defined will be a superuser, who can override all access restrictions within the database. Superuser status is dangerous and should be used only when really needed. You must yourself be a superuser to create a new superuser. NOSUPERUSER is the default.

CREATEDB
NOCREATEDB

If CREATEDB is specified, the role being defined will be allowed to create new databases. NOCREATEDB (the default) will deny a role the ability to create databases.

CREATEROLE
NOCREATEROLE

If CREATEROLE is specified, the role being defined will be allowed to create new roles, alter other roles, and drop other roles. NOCREATEROLE (the default) will deny a role the ability to create roles or modify roles other than their own.

CREATEUSER

NOCREATEUSER

These clauses are obsolete, but still accepted, spellings of SUPERUSER and NOSUPERUSER. Note that they are not equivalent to the CREATEROLE and NOCREATEROLE clauses.

CREATEEXTTABLE
NOCREATEEXTTABLE

If CREATEEXTTABLE is specified, the role being defined is allowed to create external tables. The default type is readable and the default protocol is gpfdist, if not specified. Valid types are gpfdist, gpfdists, http, and https. NOCREATEEXTTABLE (the default type) denies the role the ability to create external tables. Note that external tables that use the file or execute protocols can only be created by superusers.

Use the GRANT...ON PROTOCOL command to allow users to create and use external tables with a custom protocol type, including the s3 and pxf protocols included with Greenplum Database.

INHERIT
NOINHERIT

If specified, INHERIT (the default) allows the role to use whatever database privileges have been granted to all roles it is directly or indirectly a member of. With NOINHERIT, membership in another role only grants the ability to SET ROLE to that other role.

LOGIN
NOLOGIN

If specified, LOGIN allows a role to log in to a database. A role having the LOGIN attribute can be thought of as a user. Roles with NOLOGIN are useful for managing database privileges, and can be thought of as groups. If not specified, NOLOGIN is the default, except when CREATE ROLE is invoked through its alternative spelling *CREATE USER*.

REPLICATION
NOREPLICATION

These clauses determine whether a role is allowed to initiate streaming replication or put the system in and out of backup mode. A role having the REPLICATION attribute is a very highly privileged role, and should only be used on roles actually used for replication. If not specified, NOREPLICATION is the default .

CONNECTION LIMIT *connlimit*

The number maximum of concurrent connections this role can make. The default of -1 means there is no limitation.

PASSWORD *password*

Sets the user password for roles with the LOGIN attribute. If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as PASSWORD NULL.

ENCRYPTED
UNENCRYPTED

These key words control whether the password is stored encrypted in the system catalogs. (If neither is specified, the default behavior is determined by the configuration parameter *password_encryption*.) If the presented password string is already in MD5-encrypted format, then it is stored encrypted as-is, regardless of whether ENCRYPTED or UNENCRYPTED is specified (since the system cannot decrypt the specified encrypted password string). This allows reloading of encrypted passwords during dump/restore.

VALID UNTIL *'timestamp'*

The VALID UNTIL clause sets a date and time after which the role's password is no longer valid. If this clause is omitted the password will never expire.

IN ROLE *rolename*

Adds the new role as a member of the named roles. Note that there is no option to add the new role as an administrator; use a separate `GRANT` command to do that.

ROLE *rolename*

Adds the named roles as members of this role, making this new role a group.

ADMIN *rolename*

The `ADMIN` clause is like `ROLE`, but the named roles are added to the new role `WITH ADMIN OPTION`, giving them the right to grant membership in this role to others.

RESOURCE GROUP *group_name*

The name of the resource group to assign to the new role. The role will be subject to the concurrent transaction, memory, and CPU limits configured for the resource group. You can assign a single resource group to one or more roles.

If you do not specify a resource group for a new role, the role is automatically assigned the default resource group for the role's capability, `admin_group` for `SUPERUSER` roles, `default_group` for non-admin roles.

You can assign the `admin_group` resource group to any role having the `SUPERUSER` attribute.

You can assign the `default_group` resource group to any role.

You cannot assign a resource group that you create for an external component to a role.

RESOURCE QUEUE *queue_name*

The name of the resource queue to which the new user-level role is to be assigned. Only roles with `LOGIN` privilege can be assigned to a resource queue. The special keyword `NONE` means that the role is assigned to the default resource queue. A role can only belong to one resource queue.

Roles with the `SUPERUSER` attribute are exempt from resource queue limits. For a superuser role, queries always run immediately regardless of limits imposed by an assigned resource queue.

DENY *deny_point*
DENY BETWEEN *deny_point* AND *deny_point*

The `DENY` and `DENY BETWEEN` keywords set time-based constraints that are enforced at login. `DENY` sets a day or a day and time to deny access. `DENY BETWEEN` sets an interval during which access is denied. Both use the parameter *deny_point* that has the following format:

```
DAY day [ TIME 'time' ]
```

The two parts of the `deny_point` parameter use the following formats:

For `day`:

```
{'Sunday' | 'Monday' | 'Tuesday' |'Wednesday' | 'Thursday' |
 'Friday' |
'Saturday' | 0-6 }
```

For `time`:

```
{ 00-23 : 00-59 | 01-12 : 00-59 { AM | PM }}
```

The `DENY BETWEEN` clause uses two *deny_point* parameters:

```
DENY BETWEEN deny_point AND deny_point
```

For more information and examples about time-based constraints, see "Managing Roles and Privileges" in the *Greenplum Database Administrator Guide*.

Notes

The preferred way to add and remove role members (manage groups) is to use *GRANT* and *REVOKE*.

The VALID UNTIL clause defines an expiration time for a password only, not for the role. The expiration time is not enforced when logging in using a non-password-based authentication method.

The INHERIT attribute governs inheritance of grantable privileges (access privileges for database objects and role memberships). It does not apply to the special role attributes set by CREATE ROLE and ALTER ROLE. For example, being a member of a role with CREATEDB privilege does not immediately grant the ability to create databases, even if INHERIT is set. These privileges/attributes are never inherited: SUPERUSER, CREATEDB, CREATEROLE, CREATEEXTTABLE, LOGIN, RESOURCE GROUP, and RESOURCE QUEUE. The attributes must be set on each user-level role.

The INHERIT attribute is the default for reasons of backwards compatibility. In prior releases of Greenplum Database, users always had access to all privileges of groups they were members of. However, NOINHERIT provides a closer match to the semantics specified in the SQL standard.

Be careful with the CREATEROLE privilege. There is no concept of inheritance for the privileges of a CREATEROLE-role. That means that even if a role does not have a certain privilege but is allowed to create other roles, it can easily create another role with different privileges than its own (except for creating roles with superuser privileges). For example, if a role has the CREATEROLE privilege but not the CREATEDB privilege, it can create a new role with the CREATEDB privilege. Therefore, regard roles that have the CREATEROLE privilege as almost-superuser-roles.

The CONNECTION LIMIT option is never enforced for superusers.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in clear-text, and it might also be logged in the client's command history or the server log. The client program createuser, however, transmits the password encrypted. Also, psql contains a command \password that can be used to safely change the password later.

Examples

Create a role that can log in, but don't give it a password:

```
CREATE ROLE jonathan LOGIN;
```

Create a role that belongs to a resource queue:

```
CREATE ROLE jonathan LOGIN RESOURCE QUEUE poweruser;
```

Create a role with a password that is valid until the end of 2016 (CREATE USER is the same as CREATE ROLE except that it implies LOGIN):

```
CREATE USER joelle WITH PASSWORD 'jw8s0F4' VALID UNTIL '2017-01-01';
```

Create a role that can create databases and manage other roles:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

Create a role that does not allow login access on Sundays:

```
CREATE ROLE user3 DENY DAY 'Sunday';
```

Create a role that can create readable and writable external tables of type 'gpfdist':

```
CREATE ROLE jan WITH CREATEEXTTABLE(type='readable', protocol='gpfdist')
   CREATEEXTTABLE(type='writable', protocol='gpfdist');
```

Create a role, assigning a resource group:

```
CREATE ROLE bill RESOURCE GROUP rg_light;
```

Compatibility

The SQL standard defines the concepts of users and roles, but it regards them as distinct concepts and leaves all commands defining users to be specified by the database implementation. In Greenplum Database users and roles are unified into a single type of object. Roles therefore have many more optional attributes than they do in the standard.

`CREATE ROLE` is in the SQL standard, but the standard only requires the syntax:

```
CREATE ROLE name [WITH ADMIN rolename]
```

Allowing multiple initial administrators, and all the other options of `CREATE ROLE`, are Greenplum Database extensions.

The behavior specified by the SQL standard is most closely approximated by giving users the `NOINHERIT` attribute, while roles are given the `INHERIT` attribute.

See Also

SET ROLE, *ALTER ROLE*, *DROP ROLE*, *GRANT*, *REVOKE*, *CREATE RESOURCE QUEUE* *CREATE RESOURCE GROUP*

CREATE RULE

Defines a new rewrite rule.

Synopsis

```
CREATE [OR REPLACE] RULE name AS ON event
  TO table_name [WHERE condition]
  DO [ALSO | INSTEAD] { NOTHING | command | (command; command
  ...) }
```

Description

`CREATE RULE` defines a new rule applying to a specified table or view. `CREATE OR REPLACE RULE` will either create a new rule, or replace an existing rule of the same name for the same table.

The Greenplum Database rule system allows one to define an alternate action to be performed on insertions, updates, or deletions in database tables. A rule causes additional or alternate commands to be executed when a given command on a given table is executed. An `INSTEAD` rule can replace a given command by another, or cause a command to not be executed at all. Rules can be used to implement SQL views as well. It is important to realize that a rule is really a command transformation mechanism, or command macro. The transformation happens before the execution of the command starts. It does not operate independently for each physical row as does a trigger.

`ON SELECT` rules must be unconditional `INSTEAD` rules and must have actions that consist of a single `SELECT` command. Thus, an `ON SELECT` rule effectively turns the table into a view, whose visible contents are the rows returned by the rule's `SELECT` command rather than whatever had been stored in the table (if

anything). It is considered better style to write a CREATE VIEW command than to create a real table and define an ON SELECT rule for it.

You can create the illusion of an updatable view by defining ON INSERT, ON UPDATE, and ON DELETE rules to replace update actions on the view with appropriate updates on other tables. If you want to support INSERT RETURNING and so on, be sure to put a suitable RETURNING clause into each of these rules.

There is a catch if you try to use conditional rules for view updates: there must be an unconditional INSTEAD rule for each action you wish to allow on the view. If the rule is conditional, or is not INSTEAD, then the system will still reject attempts to perform the update action, because it thinks it might end up trying to perform the action on the dummy table of the view in some cases. If you want to handle all the useful cases in conditional rules, add an unconditional DO INSTEAD NOTHING rule to ensure that the system understands it will never be called on to update the dummy table. Then make the conditional rules non-INSTEAD; in the cases where they are applied, they add to the default INSTEAD NOTHING action. (This method does not currently work to support RETURNING queries, however.)

Note:

A view that is simple enough to be automatically updatable (see *CREATE VIEW*) does not require a user-created rule in order to be updatable. While you can create an explicit rule anyway, the automatic update transformation will generally outperform an explicit rule.

Parameters

name

The name of a rule to create. This must be distinct from the name of any other rule for the same table. Multiple rules on the same table and same event type are applied in alphabetical name order.

event

The event is one of SELECT, INSERT, UPDATE, or DELETE.

table_name

The name (optionally schema-qualified) of the table or view the rule applies to.

condition

Any SQL conditional expression (returning boolean). The condition expression may not refer to any tables except NEW and OLD, and may not contain aggregate functions. NEW and OLD refer to values in the referenced table. NEW is valid in ON INSERT and ON UPDATE rules to refer to the new row being inserted or updated. OLD is valid in ON UPDATE and ON DELETE rules to refer to the existing row being updated or deleted.

INSTEAD

INSTEAD NOTHING indicates that the commands should be executed instead of the original command.

ALSO

ALSO indicates that the commands should be executed in addition to the original command. If neither ALSO nor INSTEAD is specified, ALSO is the default.

command

The command or commands that make up the rule action. Valid commands are SELECT, INSERT, UPDATE, or DELETE. The special table names NEW and OLD may be used to refer to values in the referenced table. NEW is valid in ON INSERT and ONUPDATE rules to refer to the new row being inserted or updated. OLD is valid in ON UPDATE and ON DELETE rules to refer to the existing row being updated or deleted.

Notes

You must be the owner of a table to create or change rules for it.

It is very important to take care to avoid circular rules. Recursive rules are not validated at rule create time, but will report an error at execution time.

Examples

Create a rule that inserts rows into the child table `b2001` when a user tries to insert into the partitioned parent table `rank`:

```
CREATE RULE b2001 AS ON INSERT TO rank WHERE gender='M' and
year='2001' DO INSTEAD INSERT INTO b2001 VALUES (NEW.id,
NEW.rank, NEW.year, NEW.gender, NEW.count);
```

Compatibility

`CREATE RULE` is a Greenplum Database language extension, as is the entire query rewrite system.

See Also

ALTER RULE, *DROP RULE*, *CREATE TABLE*, *CREATE VIEW*

CREATE SCHEMA

Defines a new schema.

Synopsis

```
CREATE SCHEMA schema_name [AUTHORIZATION username]
   [schema_element [ ... ]]

CREATE SCHEMA AUTHORIZATION rolename [schema_element [ ... ]]

CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION user_name ]

CREATE SCHEMA IF NOT EXISTS AUTHORIZATION user_name
```

Description

`CREATE SCHEMA` enters a new schema into the current database. The schema name must be distinct from the name of any existing schema in the current database.

A schema is essentially a namespace: it contains named objects (tables, data types, functions, and operators) whose names may duplicate those of other objects existing in other schemas. Named objects are accessed either by qualifying their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). A `CREATE` command specifying an unqualified object name creates the object in the current schema (the one at the front of the search path, which can be determined with the function `current_schema`).

Optionally, `CREATE SCHEMA` can include subcommands to create objects within the new schema. The subcommands are treated essentially the same as separate commands issued after creating the schema, except that if the `AUTHORIZATION` clause is used, all the created objects will be owned by that role.

Parameters

schema_name

The name of a schema to be created. If this is omitted, the user name is used as the schema name. The name cannot begin with `pg_`, as such names are reserved for system catalog schemas.

user_name

The name of the role who will own the schema. If omitted, defaults to the role executing the command. Only superusers may create schemas owned by roles other than themselves.

schema_element

An SQL statement defining an object to be created within the schema. Currently, only `CREATE TABLE`, `CREATE VIEW`, `CREATE INDEX`, `CREATE SEQUENCE`, `CREATE TRIGGER` and `GRANT` are accepted as clauses within `CREATE SCHEMA`. Other kinds of objects may be created in separate commands after the schema is created.

Note: Greenplum Database does not support triggers.

`IF NOT EXISTS`

Do nothing (except issuing a notice) if a schema with the same name already exists. *schema_element* subcommands cannot be included when this option is used.

Notes

To create a schema, the invoking user must have the `CREATE` privilege for the current database or be a superuser.

Examples

Create a schema:

```
CREATE SCHEMA myschema;
```

Create a schema for role `joe` (the schema will also be named `joe`):

```
CREATE SCHEMA AUTHORIZATION joe;
```

Create a schema named `test` that will be owned by user `joe`, unless there already is a schema named `test`. (It does not matter whether `joe` owns the pre-existing schema.)

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

Compatibility

The SQL standard allows a `DEFAULT CHARACTER SET` clause in `CREATE SCHEMA`, as well as more subcommand types than are presently accepted by Greenplum Database.

The SQL standard specifies that the subcommands in `CREATE SCHEMA` may appear in any order. The present Greenplum Database implementation does not handle all cases of forward references in subcommands; it may sometimes be necessary to reorder the subcommands in order to avoid forward references.

According to the SQL standard, the owner of a schema always owns all objects within it. Greenplum Database allows schemas to contain objects owned by users other than the schema owner. This can happen only if the schema owner grants the `CREATE` privilege on the schema to someone else, or a superuser chooses to create objects in it.

The `IF NOT EXISTS` option is a Greenplum Database extension.

See Also

`ALTER SCHEMA`, *`DROP SCHEMA`*

CREATE SEQUENCE

Defines a new sequence generator.

Synopsis

```
CREATE [TEMPORARY | TEMP] SEQUENCE name
       [INCREMENT [BY] value]
       [MINVALUE minvalue | NO MINVALUE]
       [MAXVALUE maxvalue | NO MAXVALUE]
       [START [ WITH ] start]
       [CACHE cache]
       [[NO] CYCLE]
       [OWNED BY { table.column | NONE }]
```

Description

`CREATE SEQUENCE` creates a new sequence number generator. This involves creating and initializing a new special single-row table. The generator will be owned by the user issuing the command.

If a schema name is given, then the sequence is created in the specified schema. Otherwise it is created in the current schema. Temporary sequences exist in a special schema, so a schema name may not be given when creating a temporary sequence. The sequence name must be distinct from the name of any other sequence, table, index, view, or foreign table in the same schema.

After a sequence is created, you use the `nextval()` function to operate on the sequence. For example, to insert a row into a table that gets the next value of a sequence:

```
INSERT INTO distributors VALUES (nextval('myserial'), 'acme');
```

You can also use the function `setval()` to operate on a sequence, but only for queries that do not operate on distributed data. For example, the following query is allowed because it resets the sequence counter value for the sequence generator process on the master:

```
SELECT setval('myserial', 201);
```

But the following query will be rejected in Greenplum Database because it operates on distributed data:

```
INSERT INTO product VALUES (setval('myserial', 201), 'gizmo');
```

In a regular (non-distributed) database, functions that operate on the sequence go to the local sequence table to get values as they are needed. In Greenplum Database, however, keep in mind that each segment is its own distinct database process. Therefore the segments need a single point of truth to go for sequence values so that all segments get incremented correctly and the sequence moves forward in the right order. A sequence server process runs on the master and is the point-of-truth for a sequence in a Greenplum distributed database. Segments get sequence values at runtime from the master.

Because of this distributed sequence design, there are some limitations on the functions that operate on a sequence in Greenplum Database:

- `lastval()` and `currval()` functions are not supported.
- `setval()` can only be used to set the value of the sequence generator on the master, it cannot be used in subqueries to update records on distributed table data.
- `nextval()` sometimes grabs a block of values from the master for a segment to use, depending on the query. So values may sometimes be skipped in the sequence if all of the block turns out not to be needed at the segment level. Note that a regular PostgreSQL database does this too, so this is not something unique to Greenplum Database.

Although you cannot update a sequence directly, you can use a query like:

```
SELECT * FROM sequence_name;
```

to examine the parameters and current state of a sequence. In particular, the *last_value* field of the sequence shows the last value allocated by any session.

Parameters

TEMPORARY | TEMP

If specified, the sequence object is created only for this session, and is automatically dropped on session exit. Existing permanent sequences with the same name are not visible (in this session) while the temporary sequence exists, unless they are referenced with schema-qualified names.

name

The name (optionally schema-qualified) of the sequence to be created.

increment

Specifies which value is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

minvalue
NO MINVALUE

Determines the minimum value a sequence can generate. If this clause is not supplied or `NO MINVALUE` is specified, then defaults will be used. The defaults are 1 and -263-1 for ascending and descending sequences, respectively.

maxvalue
NO MAXVALUE

Determines the maximum value for the sequence. If this clause is not supplied or `NO MAXVALUE` is specified, then default values will be used. The defaults are 263-1 and -1 for ascending and descending sequences, respectively.

start

Allows the sequence to begin anywhere. The default starting value is *minvalue* for ascending sequences and *maxvalue* for descending ones.

cache

Specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum (and default) value is 1 (no cache).

CYCLE
NO CYCLE

Allows the sequence to wrap around when the *maxvalue* (for ascending) or *minvalue* (for descending) has been reached. If the limit is reached, the next number generated will be the *minvalue* (for ascending) or *maxvalue* (for descending). If `NO CYCLE` is specified, any calls to `nextval()` after the sequence has reached its maximum value will return an error. If not specified, `NO CYCLE` is the default.

OWNED BY *table.column*
OWNED BY NONE

Causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. The specified table must have the same owner and be in the same schema as the sequence. `OWNED BY NONE`, the default, specifies that there is no such association.

Notes

Sequences are based on bigint arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807).

Although multiple sessions are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the sessions are considered. For example, session A might reserve values 1..10 and return `nextval=1`, then session B might reserve values 11..20 and return `nextval=11` before session A has generated nextval=2. Thus, you should only assume that the `nextval()` values are all distinct, not that they are generated purely sequentially. Also, *last_value* will reflect the latest value reserved by any session, whether or not it has yet been returned by `nextval()`.

Examples

Create a sequence named *myseq*:

```
CREATE SEQUENCE myseq START 101;
```

Insert a row into a table that gets the next value of the sequence named *idseq*:

```
INSERT INTO distributors VALUES (nextval('idseq'), 'acme');
```

Reset the sequence counter value on the master:

```
SELECT setval('myseq', 201);
```

Illegal use of `setval()` in Greenplum Database (setting sequence values on distributed data):

```
INSERT INTO product VALUES (setval('myseq', 201), 'gizmo');
```

Compatibility

`CREATE SEQUENCE` conforms to the SQL standard, with the following exceptions:

- The `AS` *`data_type`* expression specified in the SQL standard is not supported.
- Obtaining the next value is done using the `nextval()` function instead of the `NEXT VALUE FOR` expression specified in the SQL standard.
- The `OWNED BY` clause is a Greenplum Database extension.

See Also

`ALTER SEQUENCE`, *`DROP SEQUENCE`*

CREATE SERVER

Defines a new foreign server.

Synopsis

```
CREATE SERVER server_name [ TYPE 'server_type' ] [ VERSION
 'server_version' ]
    FOREIGN DATA WRAPPER fdw_name
    [ OPTIONS ( [ mpp_execute { 'master' | 'any' | 'all segments' }
 [, ] ] option 'value' [, ... ] ) ]
```

Description

`CREATE SERVER` defines a new foreign server. The user who defines the server becomes its owner.

A foreign server typically encapsulates connection information that a foreign-data wrapper uses to access an external data source. Additional user-specific connection information may be specified by means of user mappings.

Creating a server requires the USAGE privilege on the foreign-data wrapper specified.

Parameters

server_name

The name of the foreign server to create. The server name must be unique within the database.

server_type

Optional server type, potentially useful to foreign-data wrappers.

server_version

Optional server version, potentially useful to foreign-data wrappers.

fdw_name

Name of the foreign-data wrapper that manages the server.

OPTIONS (*option* '*value*' [, ...])

The options for the new foreign server. The options typically define the connection details of the server, but the actual names and values are dependent upon the server's foreign-data wrapper.

mpp_execute { 'master' | 'any' | 'all segments' }

An option that identifies the host from which the foreign data-wrapper requests data:

- `master` (the default)—Request data from the master host.
- `any`—Request data from either the master host or any one segment, depending on which path costs less.
- `all segments`—Request data from all segments. To support this option value, the foreign data-wrapper must have a policy that matches the segments to data.

The `mpp_execute` option can be specified in multiple commands: `CREATE FOREIGN TABLE`, `CREATE SERVER`, and `CREATE FOREIGN DATA WRAPPER`. The foreign table setting takes precedence over the foreign server setting, followed by the foreign data wrapper setting.

Notes

When using the dblink module (see *dblink*), you can use the foreign server name as an argument of the `dblink_connect()` function to provide the connection parameters. You must have the USAGE privilege on the foreign server to use it in this manner.

Examples

Create a foreign server named `myserver` that uses the foreign-data wrapper named `pgsql` and includes connection options:

```
CREATE SERVER myserver FOREIGN DATA WRAPPER pgsql
    OPTIONS (host 'foo', dbname 'foodb', port '5432');
```

Compatibility

`CREATE SERVER` conforms to ISO/IEC 9075-9 (SQL/MED).

See Also

ALTER SERVER, *DROP SERVER*, *CREATE FOREIGN DATA WRAPPER*, *CREATE USER MAPPING*

CREATE TABLE

Defines a new table.

Note: Referential integrity syntax (foreign key constraints) is accepted but not enforced.

Synopsis

```
CREATE [ [GLOBAL | LOCAL] {TEMPORARY | TEMP } | UNLOGGED] TABLE [IF NOT
 EXISTS]
  table_name (
  [ { column_name data_type [ COLLATE collation ] [column_constraint
 [ ... ] ]
[ ENCODING ( storage_directive [, ...] ) ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    | [ column_reference_storage_directive [, ...]
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [=value] [, ... ] ) ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
[ DISTRIBUTED BY (column [opclass], [ ... ] )
       | DISTRIBUTED RANDOMLY | DISTRIBUTED REPLICATED ]

{ --partitioned table using SUBPARTITION TEMPLATE
[ PARTITION BY partition_type (column)
  {  [ SUBPARTITION BY partition_type (column1)
       SUBPARTITION TEMPLATE ( template_spec ) ]
          [ SUBPARTITION BY partition_type (column2)
            SUBPARTITION TEMPLATE ( template_spec ) ]
              [...]  }
  ( partition_spec ) ]
} |

{ -- partitioned table without SUBPARTITION TEMPLATE
[ PARTITION BY partition_type (column)
   [ SUBPARTITION BY partition_type (column1) ]
      [ SUBPARTITION BY partition_type (column2) ]
         [...]
  ( partition_spec
     [ ( subpartition_spec_column1
          [ ( subpartition_spec_column2
               [...] ) ] ) ],
  [ partition_spec
     [ ( subpartition_spec_column1
        [ ( subpartition_spec_column2
             [...] ) ] ) ], ]
    [...]
  ) ]
}

CREATE [ [GLOBAL | LOCAL] {TEMPORARY | TEMP} | UNLOGGED ] TABLE [IF NOT
 EXISTS]
   table_name
    OF type_name [ (
  { column_name WITH OPTIONS [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ]
```

```
[ WITH ( storage_parameter [=value] [, ... ] ) ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

where *column_constraint* is:

```
[ CONSTRAINT constraint_name]
{ NOT NULL
  | NULL
  | CHECK  ( expression ) [ NO INHERIT ]
  | DEFAULT default_expr
  | UNIQUE index_parameters
  | PRIMARY KEY index_parameters
  | REFERENCES reftable [ ( refcolumn ) ]
     [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
     [ ON DELETE key_action ] [ ON UPDATE key_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

and *table_constraint* is:

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ]
  | UNIQUE ( column_name [, ... ] ) index_parameters
  | PRIMARY KEY ( column_name [, ... ] ) index_parameters
  | FOREIGN KEY ( column_name [, ... ] )
     REFERENCES reftable [ ( refcolumn [, ... ] ) ]
     [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
     [ ON DELETE key_action ] [ ON UPDATE key_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

and *like_option* is:

```
{INCLUDING|EXCLUDING} {DEFAULTS|CONSTRAINTS|INDEXES|STORAGE|COMMENTS|ALL}
```

and *index_parameters* in `UNIQUE` and `PRIMARY KEY` constraints are:

```
[ WITH ( storage_parameter [=value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

and *storage_directive* for a column is:

```
   compresstype={ZLIB|ZSTD|QUICKLZ|RLE_TYPE|NONE}
    [compresslevel={0-9}]
    [blocksize={8192-2097152} ]
```

and *storage_parameter* for the table is:

```
   appendoptimized={TRUE|FALSE}
   blocksize={8192-2097152}
   orientation={COLUMN|ROW}
   checksum={TRUE|FALSE}
   compresstype={ZLIB|ZSTD|QUICKLZ|RLE_TYPE|NONE}
   compresslevel={0-9}
   fillfactor={10-100}
   [oids=FALSE]
```

and *key_action* is:

```
    ON DELETE
  | ON UPDATE
  | NO ACTION
```

```
   | RESTRICT
   | CASCADE
   | SET NULL
   | SET DEFAULT
```

and *partition_type* is:

```
    LIST | RANGE
```

and *partition_specification* is:

```
partition_element [, ...]
```

and *partition_element* is:

```
   DEFAULT PARTITION name
  | [PARTITION name] VALUES (list_value [,...] )
  | [PARTITION name]
     START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
     [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
     [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
  | [PARTITION name]
     END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
     [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ column_reference_storage_directive [, ...] ]
[ TABLESPACE tablespace ]
```

where *subpartition_spec* or *template_spec* is:

```
subpartition_element [, ...]
```

and *subpartition_element* is:

```
   DEFAULT SUBPARTITION name
  | [SUBPARTITION name] VALUES (list_value [,...] )
  | [SUBPARTITION name]
     START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
     [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
     [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
  | [SUBPARTITION name]
     END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
     [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ column_reference_storage_directive [, ...] ]
[ TABLESPACE tablespace ]
```

where *storage_parameter* for a partition is:

```
   appendoptimized={TRUE|FALSE}
   blocksize={8192-2097152}
   orientation={COLUMN|ROW}
   checksum={TRUE|FALSE}
   compresstype={ZLIB|ZSTD|QUICKLZ|RLE_TYPE|NONE}
   compresslevel={1-19}
   fillfactor={10-100}
   [oids=FALSE]
```

Description

`CREATE TABLE` creates an initially empty table in the current database. The user who issues the command owns the table.

To be able to create a table, you must have `USAGE` privilege on all column types or the type in the `OF` clause, respectively.

If you specify a schema name, Greenplum creates the table in the specified schema. Otherwise Greenplum creates the table in the current schema. Temporary tables exist in a special schema, so you cannot specify a schema name when creating a temporary table. Table names must be distinct from the name of any other table, external table, sequence, index, view, or foreign table in the same schema.

`CREATE TABLE` also automatically creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

The optional constraint clauses specify conditions that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways. Constraints apply to tables, not to partitions. You cannot add a constraint to a partition or subpartition.

Referential integrity constraints (foreign keys) are accepted but not enforced. The information is kept in the system catalogs but is otherwise ignored.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience for use when the constraint only affects one column.

When creating a table, there is an additional clause to declare the Greenplum Database distribution policy. If a `DISTRIBUTED BY`, `DISTRIBUTED RANDOMLY`, or `DISTRIBUTED REPLICATED` clause is not supplied, then Greenplum Database assigns a hash distribution policy to the table using either the `PRIMARY KEY` (if the table has one) or the first column of the table as the distribution key. Columns of geometric or user-defined data types are not eligible as Greenplum distribution key columns. If a table does not have a column of an eligible data type, the rows are distributed based on a round-robin or random distribution. To ensure an even distribution of data in your Greenplum Database system, you want to choose a distribution key that is unique for each record, or if that is not possible, then choose `DISTRIBUTED RANDOMLY`.

If the `DISTRIBUTED REPLICATED` clause is supplied, Greenplum Database distributes all rows of the table to all segments in the Greenplum Database system. This option can be used in cases where user-defined functions must execute on the segments, and the functions require access to all rows of the table. Replicated functions can also be used to improve query performance by preventing broadcast motions for the table. The `DISTRIBUTED REPLICATED` clause cannot be used with the `PARTITION BY` clause or the `INHERITS` clause. A replicated table also cannot be inherited by another table. The hidden system columns (`ctid`, `cmin`, `cmax`, `xmin`, `xmax`, and `gp_segment_id`) cannot be referenced in user queries on replicated tables because they have no single, unambiguous value. Greenplum Database returns a `column does not exist` error for the query.

The `PARTITION BY` clause allows you to divide the table into multiple sub-tables (or parts) that, taken together, make up the parent table and share its schema. Though the sub-tables exist as independent tables, the Greenplum Database restricts their use in important ways. Internally, partitioning is implemented as a special form of inheritance. Each child table partition is created with a distinct `CHECK` constraint which limits the data the table can contain, based on some defining criteria. The `CHECK` constraints are also used by the query optimizer to determine which table partitions to scan in order to satisfy a given query predicate. These partition constraints are managed automatically by the Greenplum Database.

Parameters

`GLOBAL | LOCAL`

These keywords are present for SQL standard compatibility, but have no effect in Greenplum Database and are deprecated.

TEMPORARY | TEMP

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see `ON COMMIT`). Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names. Any indexes created on a temporary table are automatically temporary as well.

UNLOGGED

If specified, the table is created as an unlogged table. Data written to unlogged tables is not written to the write-ahead (WAL) log, which makes them considerably faster than ordinary tables. However, the contents of an unlogged table are not replicated to mirror segment instances. Also an unlogged table is not crash-safe. After a segment instance crash or unclean shutdown, the data for the unlogged table on that segment is truncated. Any indexes created on an unlogged table are automatically unlogged as well.

table_name

The name (optionally schema-qualified) of the table to be created.

OF *type_name*

Creates a typed table, which takes its structure from the specified composite type (name optionally schema-qualified). A typed table is tied to its type; for example the table will be dropped if the type is dropped (with `DROP TYPE ... CASCADE`).

When a typed table is created, the data types of the columns are determined by the underlying composite type and are not specified by the `CREATE TABLE` command. But the `CREATE TABLE` command can add defaults and constraints to the table and can specify storage parameters.

column_name

The name of a column to be created in the new table.

data_type

The data type of the column. This may include array specifiers.

For table columns that contain textual data, Specify the data type `VARCHAR` or `TEXT`. Specifying the data type `CHAR` is not recommended. In Greenplum Database, the data types `VARCHAR` or `TEXT` handles padding added to the data (space characters added after the last non-space character) as significant characters, the data type `CHAR` does not. See *Notes*.

COLLATE *collation*

The `COLLATE` clause assigns a collation to the column (which must be of a collatable data type). If not specified, the column data type's default collation is used.

> **Note:** GPORCA supports collation only when all columns in the query use the same collation. If columns in the query use different collations, then Greenplum uses the Postgres Planner.

DEFAULT *default_expr*

The `DEFAULT` clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column. The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

ENCODING (*storage_directive* [, ...])

For a column, the optional `ENCODING` clause specifies the type of compression and block size for the column data. See *storage_options* for `compresstype`, `compresslevel`, and `blocksize` values.

The clause is valid only for append-optimized, column-oriented tables.

Column compression settings are inherited from the table level to the partition level to the subpartition level. The lowest-level settings have priority.

`INHERITS ( parent_table [, …])`

The optional `INHERITS` clause specifies a list of tables from which the new table automatically inherits all columns. Use of `INHERITS` creates a persistent relationship between the new child table and its parent table(s). Schema modifications to the parent(s) normally propagate to children as well, and by default the data of the child table is included in scans of the parent(s).

In Greenplum Database, the `INHERITS` clause is not used when creating partitioned tables. Although the concept of inheritance is used in partition hierarchies, the inheritance structure of a partitioned table is created using the *`PARTITION BY`* clause.

If the same column name exists in more than one parent table, an error is reported unless the data types of the columns match in each of the parent tables. If there is no conflict, then the duplicate columns are merged to form a single column in the new table. If the column name list of the new table contains a column name that is also inherited, the data type must likewise match the inherited column(s), and the column definitions are merged into one. If the new table explicitly specifies a default value for the column, this default overrides any defaults from inherited declarations of the column. Otherwise, any parents that specify default values for the column must all specify the same default, or an error will be reported.

`CHECK` constraints are merged in essentially the same way as columns: if multiple parent tables or the new table definition contain identically-named `constraints`, these constraints must all have the same check expression, or an error will be reported. Constraints having the same name and expression will be merged into one copy. A constraint marked `NO INHERIT` in a parent will not be considered. Notice that an unnamed `CHECK` constraint in the new table will never be merged, since a unique name will always be chosen for it.

Column `STORAGE` settings are also copied from parent tables.

`LIKE *source_table like_option* ...]`

The `LIKE` clause specifies a table from which the new table automatically copies all column names, their data types, not-null constraints, and distribution policy. Storage properties like append-optimized or partition structure are not copied. Unlike `INHERITS`, the new table and original table are completely decoupled after creation is complete.

Default expressions for the copied column definitions will only be copied if `INCLUDING DEFAULTS` is specified. The default behavior is to exclude default expressions, resulting in the copied columns in the new table having null defaults.

Not-null constraints are always copied to the new table. `CHECK` constraints will be copied only if `INCLUDING CONSTRAINTS` is specified. No distinction is made between column constraints and table constraints.

Indexes, `PRIMARY KEY`, and `UNIQUE` constraints on the original table will be created on the new table only if the `INCLUDING INDEXES` clause is specified. Names for the new indexes and constraints are chosen according to the default rules, regardless of how the originals were named. (This behavior avoids possible duplicate-name failures for the new indexes.)

Any indexes on the original table will not be created on the new table, unless the `INCLUDING INDEXES` clause is specified.

STORAGE settings for the copied column definitions will be copied only if INCLUDING STORAGE is specified. The default behavior is to exclude STORAGE settings, resulting in the copied columns in the new table having type-specific default settings.

Comments for the copied columns, constraints, and indexes will be copied only if INCLUDING COMMENTS is specified. The default behavior is to exclude comments, resulting in the copied columns and constraints in the new table having no comments.

INCLUDING ALL is an abbreviated form of INCLUDING DEFAULTS INCLUDING CONSTRAINTS INCLUDING INDEXES INCLUDING STORAGE INCLUDING COMMENTS.

Note that unlike INHERITS, columns and constraints copied by LIKE are not merged with similarly named columns and constraints. If the same name is specified explicitly or in another LIKE clause, an error is signaled.

The LIKE clause can also be used to copy columns from views, foreign tables, or composite types. Inapplicable options (e.g., INCLUDING INDEXES from a view) are ignored.

CONSTRAINT *constraint_name*

An optional name for a column or table constraint. If the constraint is violated, the constraint name is present in error messages, so constraint names like *column must be positive* can be used to communicate helpful constraint information to client applications. (Double-quotes are needed to specify constraint names that contain spaces.) If a constraint name is not specified, the system generates a name.

> **Note:** The specified *constraint_name* is used for the constraint, but a system-generated unique name is used for the index name. In some prior releases, the provided name was used for both the constraint name and the index name.

NULL | NOT NULL

Specifies if the column is or is not allowed to contain null values. NULL is the default.

CHECK (*expression*) [NO INHERIT]

The CHECK clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or UNKNOWN succeed. Should any row of an insert or update operation produce a FALSE result an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column's value only, while an expression appearing in a table constraint can reference multiple columns.

A constraint marked with NO INHERIT will not propagate to child tables.

Currently, CHECK expressions cannot contain subqueries nor refer to variables other than columns of the current row.

UNIQUE (*column_constraint*)
UNIQUE (*column_name* [, ...]) (*table_constraint*)

The UNIQUE constraint specifies that a group of one or more columns of a table may contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns. For the purpose of a unique constraint, null values are not considered equal. The column(s) that are unique must contain all the columns of the Greenplum distribution key. In addition, the <key> must contain all the columns in the partition key if the table is partitioned. Note that a <key> constraint in a partitioned table is not the same as a simple UNIQUE INDEX.

For information about unique constraint management and limitations, see *Notes*.

PRIMARY KEY (*column constraint*)
PRIMARY KEY (*column_name* [, ...]) (*table constraint*)

The PRIMARY KEY constraint specifies that a column or columns of a table may contain only unique (non-duplicate), non-null values. Only one primary key can be specified for a table, whether as a column constraint or a table constraint.

For a table to have a primary key, it must be hash distributed (not randomly distributed), and the primary key, the column(s) that are unique, must contain all the columns of the Greenplum distribution key. In addition, the <key> must contain all the columns in the partition key if the table is partitioned. Note that a <key> constraint in a partitioned table is not the same as a simple UNIQUE INDEX.

PRIMARY KEY enforces the same data constraints as a combination of UNIQUE and NOT NULL, but identifying a set of columns as the primary key also provides metadata about the design of the schema, since a primary key implies that other tables can rely on this set of columns as a unique identifier for rows.

For information about primary key management and limitations, see *Notes*.

REFERENCES *reftable* [(*refcolumn*)]
[MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
[ON DELETE | ON UPDATE] [*key_action*]
FOREIGN KEY (*column_name* [, ...])

The REFERENCES and FOREIGN KEY clauses specify referential integrity constraints (foreign key constraints). Greenplum accepts referential integrity constraints as specified in PostgreSQL syntax but does not enforce them. See the PostgreSQL documentation for information about referential integrity constraints.

DEFERRABLE
NOT DEFERRABLE

The [NOT] DEFERRABLE clause controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable can be postponed until the end of the transaction (using the *SET CONSTRAINTS* command). NOT DEFERRABLE is the default. Currently, only UNIQUE and PRIMARY KEY constraints are deferrable. NOT NULL and CHECK constraints are not deferrable. REFERENCES (foreign key) constraints accept this clause but are not enforced.

INITIALLY IMMEDIATE
INITIALLY DEFERRED

If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is INITIALLY IMMEDIATE, it is checked after each statement. This is the default. If the constraint is INITIALLY DEFERRED, it is checked only at the end of the transaction. The constraint check time can be altered with the SET CONSTRAINTS command.

WITH (*storage_parameter=value*)

The WITH clause can specify storage parameters for tables, and for indexes associated with a UNIQUE or PRIMARY constraint. Note that you can also set storage parameters on a particular partition or subpartition by declaring the WITH clause in the partition specification. The lowest-level settings have priority.

The defaults for some of the table storage options can be specified with the server configuration parameter gp_default_storage_options. For information about setting default storage options, see *Notes*.

The following storage options are available:

appendoptimized — Set to TRUE to create the table as an append-optimized table. If FALSE or not declared, the table will be created as a regular heap-storage table.

blocksize — Set to the size, in bytes, for each block in a table. The blocksize must be between 8192 and 2097152 bytes, and be a multiple of 8192. The default is 32768.

orientation — Set to `column` for column-oriented storage, or `row` (the default) for row-oriented storage. This option is only valid if `appendoptimized=TRUE`. Heap-storage tables can only be row-oriented.

checksum — This option is valid only for append-optimized tables (`appendoptimized=TRUE`). The value `TRUE` is the default and enables CRC checksum validation for append-optimized tables. The checksum is calculated during block creation and is stored on disk. Checksum validation is performed during block reads. If the checksum calculated during the read does not match the stored checksum, the transaction is aborted. If you set the value to `FALSE` to disable checksum validation, checking the table data for on-disk corruption will not be performed.

compresstype — Set to `ZLIB` (the default), `ZSTD`, `RLE_TYPE`, or `QUICKLZ`[1] to specify the type of compression used. The value `NONE` disables compression. Zstd provides for both speed or a good compression ratio, tunable with the `compresslevel` option. QuickLZ and zlib are provided for backwards-compatibility. Zstd outperforms these compression types on usual workloads. The `compresstype` option is only valid if `appendoptimized=TRUE`.

> **Note:** [1]QuickLZ compression is available only in the commercial release of Pivotal Greenplum Database.

The value `RLE_TYPE`, which is supported only if `orientation=column` is specified, enables the run-length encoding (RLE) compression algorithm. RLE compresses data better than the Zstd, zlib, or QuickLZ compression algorithms when the same data value occurs in many consecutive rows.

For columns of type `BIGINT`, `INTEGER`, `DATE`, `TIME`, or `TIMESTAMP`, delta compression is also applied if the `compresstype` option is set to `RLE_TYPE` compression. The delta compression algorithm is based on the delta between column values in consecutive rows and is designed to improve compression when data is loaded in sorted order or the compression is applied to column data that is in sorted order.

For information about using table compression, see "Choosing the Table Storage Model" in the *Greenplum Database Administrator Guide*.

compresslevel — For Zstd compression of append-optimized tables, set to an integer value from 1 (fastest compression) to 19 (highest compression ratio). For zlib compression, the valid range is from 1 to 9. QuickLZ compression level can only be set to 1. If not declared, the default is 1. For `RLE_TYPE`, the compression level can be an integer value from 1 (fastest compression) to 4 (highest compression ratio).

The `compresslevel` option is valid only if `appendoptimized=TRUE`.

fillfactor — The fillfactor for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller fillfactor is specified, `INSERT` operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives `UPDATE` a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For a table whose entries are never updated, complete packing is the best choice, but in heavily updated tables smaller fillfactors are appropriate. This parameter cannot be set for TOAST tables.

oids=FALSE — This setting is the default, and it ensures that rows do not have object identifiers assigned to them. Pivotal does not support using `WITH OIDS` or `oids=TRUE` to assign an OID system column. On large tables, such as those in a typical Greenplum Database system, using OIDs for table rows can cause wrap-around of the 32-bit OID counter. Once the counter wraps around, OIDs can no longer be assumed to be unique, which not only makes them useless to user applications, but can also cause problems in the Greenplum Database system catalog tables. In addition, excluding OIDs from a table reduces the space required to store the table on disk by 4 bytes per row, slightly improving performance. You cannot create OIDS on a partitioned or column-oriented table (an

error is displayed). This syntax is deprecated and will be removed in a future Greenplum release.

ON COMMIT

The behavior of temporary tables at the end of a transaction block can be controlled using `ON COMMIT`. The three options are:

PRESERVE ROWS - No special action is taken at the ends of transactions for temporary tables. This is the default behavior.

DELETE ROWS - All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic `TRUNCATE` is done at each commit.

DROP - The temporary table will be dropped at the end of the current transaction block.

TABLESPACE *tablespace*

The name of the tablespace in which the new table is to be created. If not specified, the database's default tablespace is used, or *temp_tablespaces* if the table is temporary.

USING INDEX TABLESPACE *tablespace*

This clause allows selection of the tablespace in which the index associated with a `UNIQUE` or `PRIMARY KEY` constraint will be created. If not specified, the database's default tablespace is used, or *temp_tablespaces* if the table is temporary.

DISTRIBUTED BY (*column* [*opclass*], [...])
DISTRIBUTED RANDOMLY
DISTRIBUTED REPLICATED

Used to declare the Greenplum Database distribution policy for the table. `DISTRIBUTED BY` uses hash distribution with one or more columns declared as the distribution key. For the most even data distribution, the distribution key should be the primary key of the table or a unique column (or set of columns). If that is not possible, then you may choose `DISTRIBUTED RANDOMLY`, which will send the data round-robin to the segment instances. Additionally, an operator class, `opclass`, can be specified, to use a non-default hash function.

The Greenplum Database server configuration parameter `gp_create_table_random_default_distribution` controls the default table distribution policy if the `DISTRIBUTED BY` clause is not specified when you create a table. Greenplum Database follows these rules to create a table if a distribution policy is not specified.

If the value of the parameter is `off` (the default), Greenplum Database chooses the table distribution key based on the command:

- If a `LIKE` or `INHERITS` clause is specified, then Greenplum copies the distribution key from the source or parent table.
- If a `PRIMARY KEY` or `UNIQUE` constraints are specified, then Greenplum chooses the largest subset of all the key columns as the distribution key.
- If neither constraints nor a `LIKE` or `INHERITS` clause is specified, then Greenplum chooses the first suitable column as the distribution key. (Columns with geometric or user-defined data types are not eligible as Greenplum distribution key columns.)

If the value of the parameter is set to `on`, Greenplum Database follows these rules:

- If `PRIMARY KEY` or `UNIQUE` columns are not specified, the distribution of the table is random (`DISTRIBUTED RANDOMLY`). Table distribution is random even if the table creation command contains the `LIKE` or `INHERITS` clause.
- If `PRIMARY KEY` or `UNIQUE` columns are specified, a `DISTRIBUTED BY` clause must also be specified. If a `DISTRIBUTED BY` clause is not specified as part of the table creation command, the command fails.

For more information about setting the default table distribution policy, see *gp_create_table_random_default_distribution*.

The DISTRIBUTED REPLICATED clause replicates the entire table to all Greenplum Database segment instances. It can be used when it is necessary to execute user-defined functions on segments when the functions require access to all rows in the table, or to improve query performance by preventing broadcast motions.

PARTITION BY

Declares one or more columns by which to partition the table.

When creating a partitioned table, Greenplum Database creates the root partitioned table (the root partition) with the specified table name. Greenplum Database also creates a hierarchy of tables, child tables, that are the subpartitions based on the partitioning options that you specify. The Greenplum Database *pg_partition** system views contain information about the subpartition tables.

For each partition level (each hierarchy level of tables), a partitioned table can have a maximum of 32,767 partitions.

> **Note:** Greenplum Database stores partitioned table data in the leaf child tables, the lowest-level tables in the hierarchy of child tables for use by the partitioned table.

partition_type

Declares partition type: LIST (list of values) or RANGE (a numeric or date range).

partition_specification

Declares the individual partitions to create. Each partition can be defined individually or, for range partitions, you can use the EVERY clause (with a START and optional END clause) to define an increment pattern to use to create the individual partitions.

DEFAULT PARTITION *name* — Declares a default partition. When data does not match to an existing partition, it is inserted into the default partition. Partition designs that do not have a default partition will reject incoming rows that do not match to an existing partition.

PARTITION *name* — Declares a name to use for the partition. Partitions are created using the following naming convention: *parentname_level#*_prt_*givenname*.

VALUES — For list partitions, defines the value(s) that the partition will contain.

START — For range partitions, defines the starting range value for the partition. By default, start values are INCLUSIVE. For example, if you declared a start date of '2016-01-01', then the partition would contain all dates greater than or equal to '2016-01-01'. Typically the data type of the START expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

END — For range partitions, defines the ending range value for the partition. By default, end values are EXCLUSIVE. For example, if you declared an end date of '2016-02-01', then the partition would contain all dates less than but not equal to '2016-02-01'. Typically the data type of the END expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

EVERY — For range partitions, defines how to increment the values from START to END to create individual partitions. Typically the data type of the EVERY expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

WITH — Sets the table storage options for a partition. For example, you may want older partitions to be append-optimized tables and newer partitions to be regular heap tables.

TABLESPACE — The name of the tablespace in which the partition is to be created.

SUBPARTITION BY

Declares one or more columns by which to subpartition the first-level partitions of the table. The format of the subpartition specification is similar to that of a partition specification described above.

SUBPARTITION TEMPLATE

Instead of declaring each subpartition definition individually for each partition, you can optionally declare a subpartition template to be used to create the subpartitions (lower level child tables). This subpartition specification would then apply to all parent partitions.

Notes

- In Greenplum Database (a Postgres-based system) the data types `VARCHAR` or `TEXT` handle padding added to the textual data (space characters added after the last non-space character) as significant characters; the data type `CHAR` does not.

  In Greenplum Database, values of type `CHAR(`*n*`)` are padded with trailing spaces to the specified width *n*. The values are stored and displayed with the spaces. However, the padding spaces are treated as semantically insignificant. When the values are distributed, the trailing spaces are disregarded. The trailing spaces are also treated as semantically insignificant when comparing two values of data type `CHAR`, and the trailing spaces are removed when converting a character value to one of the other string types.
- Pivotal does not support using `WITH OIDS` or `oids=TRUE` to assign an OID system column. This syntax is deprecated and will be removed in a future Greenplum release. As an alternative, use a `SERIAL` or other sequence generator as the table's primary key. However, if your application does make use of OIDs to identify specific rows of a table, it is recommended to create a unique constraint on the OID column of that table, to ensure that OIDs in the table will indeed uniquely identify rows even after counter wrap-around. Avoid assuming that OIDs are unique across tables; if you need a database-wide unique identifier, use the combination of table OID and row OID for that purpose.
- Greenplum Database has some special conditions for primary key and unique constraints with regards to columns that are the *distribution key* in a Greenplum table. For a unique constraint to be enforced in Greenplum Database, the table must be hash-distributed (not `DISTRIBUTED RANDOMLY`), and the constraint columns must be the same as (or a superset of) the table's distribution key columns.

  Replicated tables (`DISTRIBUTED REPLICATED`) can have both `PRIMARY KEY` and `UNIQUE`column constraints.

  A primary key constraint is simply a combination of a unique constraint and a not-null constraint.

  Greenplum Database automatically creates a `UNIQUE` index for each `UNIQUE` or `PRIMARY KEY` constraint to enforce uniqueness. Thus, it is not necessary to create an index explicitly for primary key columns. `UNIQUE` and `PRIMARY KEY` constraints are not allowed on append-optimized tables because the `UNIQUE` indexes that are created by the constraints are not allowed on append-optimized tables.

  Foreign key constraints are not supported in Greenplum Database.

  For inherited tables, unique constraints, primary key constraints, indexes and table privileges are *not* inherited in the current implementation.
- For append-optimized tables, `UPDATE` and `DELETE` are not allowed in a repeatable read or serializable transaction and will cause the transaction to abort. `CLUSTER`, `DECLARE...FOR UPDATE`, and triggers are not supported with append-optimized tables.
- To insert data into a partitioned table, you specify the root partitioned table, the table created with the `CREATE TABLE` command. You also can specify a leaf child table of the partitioned table in an `INSERT` command. An error is returned if the data is not valid for the specified leaf child table. Specifying a child table that is not a leaf child table in the `INSERT` command is not supported. Execution of other DML commands such as `UPDATE` and `DELETE` on any child table of a partitioned table is not supported. These commands must be executed on the root partitioned table, the table created with the `CREATE TABLE` command.
- The default values for these table storage options can be specified with the server configuration parameter `gp_default_storage_option`.

- `appendoptimized`
- `blocksize`
- `checksum`
- `compresstype`
- `compresslevel`
- `orientation`

The defaults can be set for the system, a database, or a user. For information about setting storage options, see the server configuration parameter *`gp_default_storage_options`*.

Important: The current Postgres Planner allows list partitions with multi-column (composite) partition keys. GPORCA does not support composite keys, so using composite partition keys is not recommended.

Examples

Create a table named `rank` in the schema named `baby` and distribute the data using the columns `rank`, `gender`, and `year`:

```
CREATE TABLE baby.rank (id int, rank int, year smallint,
gender char(1), count int ) DISTRIBUTED BY (rank, gender,
year);
```

Create table films and table distributors (the primary key will be used as the Greenplum distribution key by default):

```
CREATE TABLE films (
code        char(5) CONSTRAINT firstkey PRIMARY KEY,
title       varchar(40) NOT NULL,
did         integer NOT NULL,
date_prod   date,
kind        varchar(10),
len         interval hour to minute
);

CREATE TABLE distributors (
did    integer PRIMARY KEY DEFAULT nextval('serial'),
name   varchar(40) NOT NULL CHECK (name <> '')
);
```

Create a gzip-compressed, append-optimized table:

```
CREATE TABLE sales (txn_id int, qty int, date date)
WITH (appendoptimized=true, compresslevel=5)
DISTRIBUTED BY (txn_id);
```

Create a simple, single level partitioned table:

```
CREATE TABLE sales (id int, year int, qtr int, c_rank int, code char(1),
 region text)
DISTRIBUTED BY (id)
PARTITION BY LIST (code)
( PARTITION sales VALUES ('S'),
  PARTITION returns VALUES ('R')
);
```

Create a three level partitioned table that defines subpartitions without the `SUBPARTITION TEMPLATE` clause:

```
CREATE TABLE sales (id int, year int, qtr int, c_rank int, code char(1),
 region text)
DISTRIBUTED BY (id)
PARTITION BY LIST (code)
  SUBPARTITION BY RANGE (c_rank)
    SUBPARTITION by LIST (region)

( PARTITION sales VALUES ('S')
   ( SUBPARTITION cr1 START (1) END (2)
      ( SUBPARTITION ca VALUES ('CA') ),
      SUBPARTITION cr2 START (3) END (4)
        ( SUBPARTITION ca VALUES ('CA') ) ),

 PARTITION returns VALUES ('R')
   ( SUBPARTITION cr1 START (1) END (2)
      ( SUBPARTITION ca VALUES ('CA') ),
     SUBPARTITION cr2 START (3) END (4)
        ( SUBPARTITION ca VALUES ('CA') ) )
);
```

Create the same partitioned table as the previous table using the `SUBPARTITION TEMPLATE` clause:

```
CREATE TABLE sales1 (id int, year int, qtr int, c_rank int, code char(1),
 region text)
DISTRIBUTED BY (id)
PARTITION BY LIST (code)

   SUBPARTITION BY RANGE (c_rank)
     SUBPARTITION TEMPLATE (
     SUBPARTITION cr1 START (1) END (2),
     SUBPARTITION cr2 START (3) END (4) )

     SUBPARTITION BY LIST (region)
       SUBPARTITION TEMPLATE (
       SUBPARTITION ca VALUES ('CA') )

( PARTITION sales VALUES ('S'),
  PARTITION  returns VALUES ('R')
) ;
```

Create a three level partitioned table using subpartition templates and default partitions at each level:

```
CREATE TABLE sales (id int, year int, qtr int, c_rank int, code char(1),
 region text)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)

  SUBPARTITION BY RANGE (qtr)
    SUBPARTITION TEMPLATE (
    START (1) END (5) EVERY (1),
    DEFAULT SUBPARTITION bad_qtr )

    SUBPARTITION BY LIST (region)
      SUBPARTITION TEMPLATE (
      SUBPARTITION usa VALUES ('usa'),
      SUBPARTITION europe VALUES ('europe'),
      SUBPARTITION asia VALUES ('asia'),
      DEFAULT SUBPARTITION other_regions)
```

```
( START (2009) END (2011) EVERY (1),
  DEFAULT PARTITION outlying_years);
```

Compatibility

`CREATE TABLE` command conforms to the SQL standard, with the following exceptions:

- **Temporary Tables** — In the SQL standard, temporary tables are defined just once and automatically exist (starting with empty contents) in every session that needs them. Greenplum Database instead requires each session to issue its own `CREATE TEMPORARY TABLE` command for each temporary table to be used. This allows different sessions to use the same temporary table name for different purposes, whereas the standard's approach constrains all instances of a given temporary table name to have the same table structure.

  The standard's distinction between global and local temporary tables is not in Greenplum Database. Greenplum Database will accept the `GLOBAL` and `LOCAL` keywords in a temporary table declaration, but they have no effect and are deprecated.

  If the `ON COMMIT` clause is omitted, the SQL standard specifies that the default behavior as `ON COMMIT DELETE ROWS`. However, the default behavior in Greenplum Database is `ON COMMIT PRESERVE ROWS`. The `ON COMMIT DROP` option does not exist in the SQL standard.
- **Column Check Constraints** — The SQL standard says that `CHECK` column constraints may only refer to the column they apply to; only `CHECK` table constraints may refer to multiple columns. Greenplum Database does not enforce this restriction; it treats column and table check constraints alike.
- **NULL Constraint** — The `NULL` constraint is a Greenplum Database extension to the SQL standard that is included for compatibility with some other database systems (and for symmetry with the `NOT NULL` constraint). Since it is the default for any column, its presence is not required.
- **Inheritance** — Multiple inheritance via the `INHERITS` clause is a Greenplum Database language extension. SQL:1999 and later define single inheritance using a different syntax and different semantics. SQL:1999-style inheritance is not yet supported by Greenplum Database.
- **Partitioning** — Table partitioning via the `PARTITION BY` clause is a Greenplum Database language extension.
- **Zero-column tables** — Greenplum Database allows a table of no columns to be created (for example, `CREATE TABLE foo();`). This is an extension from the SQL standard, which does not allow zero-column tables. Zero-column tables are not in themselves very useful, but disallowing them creates odd special cases for `ALTER TABLE DROP COLUMN`, so Greenplum decided to ignore this spec restriction.
- **LIKE** — While a `LIKE` clause exists in the SQL standard, many of the options that Greenplum Database accepts for it are not in the standard, and some of the standard's options are not implemented by Greenplum Database.
- **WITH clause** — The `WITH` clause is a Greenplum Database extension; neither storage parameters nor OIDs are in the standard.
- **Tablespaces** — The Greenplum Database concept of tablespaces is not part of the SQL standard. The clauses `TABLESPACE` and `USING INDEX TABLESPACE` are extensions.
- **Data Distribution** — The Greenplum Database concept of a parallel or distributed database is not part of the SQL standard. The `DISTRIBUTED` clauses are extensions.

See Also

`ALTER TABLE`, *`DROP TABLE`*, *`CREATE EXTERNAL TABLE`*, *`CREATE TABLE AS`*

CREATE TABLE AS

Defines a new table from the results of a query.

Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ]
 TABLE table_name
        [ (column_name [, ...] ) ]
        [ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
        [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
        [ TABLESPACE tablespace_name ]
       AS query
        [ WITH [ NO ] DATA ]
        [ DISTRIBUTED BY (column [, ... ] ) | DISTRIBUTED RANDOMLY |
 DISTRIBUTED REPLICATED ]
```

where *storage_parameter* is:

```
   appendoptimized={TRUE|FALSE}
   blocksize={8192-2097152}
   orientation={COLUMN|ROW}
   compresstype={ZLIB|ZSTD|QUICKLZ}
   compresslevel={1-19 | 1}
   fillfactor={10-100}
   [oids=FALSE]
```

Description

`CREATE TABLE AS` creates a table and fills it with data computed by a *SELECT* command. The table columns have the names and data types associated with the output columns of the `SELECT`, however you can override the column names by giving an explicit list of new column names.

`CREATE TABLE AS` creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query.

Parameters

`GLOBAL | LOCAL`

Ignored for compatibility. These keywords are deprecated; refer to *CREATE TABLE* for details.

`TEMPORARY | TEMP`

If specified, the new table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see `ON COMMIT`). Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names. Any indexes created on a temporary table are automatically temporary as well.

`UNLOGGED`

If specified, the table is created as an unlogged table. Data written to unlogged tables is not written to the write-ahead (WAL) log, which makes them considerably faster than ordinary tables. However, the contents of an unlogged table are not replicated to mirror segment instances. Also an unlogged table is not crash-safe. After a segment instance crash or unclean shutdown, the data for the unlogged table on that segment is truncated. Any indexes created on an unlogged table are automatically unlogged as well.

`table_name`

The name (optionally schema-qualified) of the new table to be created.

`column_name`

The name of a column in the new table. If column names are not provided, they are taken from the output column names of the query.

WITH (*storage_parameter=value*)

The `WITH` clause can be used to set storage options for the table or its indexes. Note that you can also set different storage parameters on a particular partition or subpartition by declaring the `WITH` clause in the partition specification. The following storage options are available:

appendoptimized — Set to `TRUE` to create the table as an append-optimized table. If `FALSE` or not declared, the table will be created as a regular heap-storage table.

blocksize — Set to the size, in bytes for each block in a table. The `blocksize` must be between 8192 and 2097152 bytes, and be a multiple of 8192. The default is 32768.

orientation — Set to `column` for column-oriented storage, or `row` (the default) for row-oriented storage. This option is only valid if `appendoptimized=TRUE`. Heap-storage tables can only be row-oriented.

compresstype — Set to `ZLIB` (the default), `ZSTD`, or `QUICKLZ`[1] to specify the type of compression used. The value `NONE` disables compression. Zstd provides for both speed or a good compression ratio, tunable with the `compresslevel` option. QuickLZ and zlib are provided for backwards-compatibility. Zstd outperforms these compression types on usual workloads. The `compresstype` option is valid only if `appendoptimized=TRUE`.

> **Note:** [1]QuickLZ compression is available only in the commercial release of Pivotal Greenplum Database.

compresslevel — For Zstd compression of append-optimized tables, set to an integer value from 1 (fastest compression) to 19 (highest compression ratio). For zlib compression, the valid range is from 1 to 9. QuickLZ compression level can only be set to 1. If not declared, the default is 1. The `compresslevel` option is valid only if `appendoptimized=TRUE`.

fillfactor — See *`CREATE INDEX`* for more information about this index storage parameter.

oids=FALSE — This setting is the default, and it ensures that rows do not have object identifiers assigned to them. Pivotal does not support using `WITH OIDS` or `oids=TRUE` to assign an OID system column. On large tables, such as those in a typical Greenplum Database system, using OIDs for table rows can cause wrap-around of the 32-bit OID counter. Once the counter wraps around, OIDs can no longer be assumed to be unique, which not only makes them useless to user applications, but can also cause problems in the Greenplum Database system catalog tables. In addition, excluding OIDs from a table reduces the space required to store the table on disk by 4 bytes per row, slightly improving performance. You cannot create OIDS on a partitioned or column-oriented table (an error is displayed). This syntax is deprecated and will be removed in a future Greenplum release.

ON COMMIT

The behavior of temporary tables at the end of a transaction block can be controlled using `ON COMMIT`. The three options are:

PRESERVE ROWS — No special action is taken at the ends of transactions for temporary tables. This is the default behavior.

DELETE ROWS — All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic `TRUNCATE` is done at each commit.

DROP — The temporary table will be dropped at the end of the current transaction block.

TABLESPACE *tablespace_name*

The *tablespace_name* parameter is the name of the tablespace in which the new table is to be created. If not specified, the database's default tablespace is used, or *temp_tablespaces* if the table is temporary.

AS *query*

A *SELECT*, *TABLE*, or *VALUES* command, or an *EXECUTE* command that runs a prepared SELECT or VALUES query.

DISTRIBUTED BY ({*column* [*opclass*]}, [...])
DISTRIBUTED RANDOMLY
DISTRIBUTED REPLICATED

Used to declare the Greenplum Database distribution policy for the table. `DISTRIBUTED BY` uses hash distribution with one or more columns declared as the distribution key. For the most even data distribution, the distribution key should be the primary key of the table or a unique column (or set of columns). If that is not possible, then you may choose `DISTRIBUTED RANDOMLY`, which will send the data round-robin to the segment instances.

`DISTRIBUTED REPLICATED` replicates all rows in the table to all Greenplum Database segments. It cannot be used with partitioned tables or with tables that inhert from other tables.

The Greenplum Database server configuration parameter `gp_create_table_random_default_distribution` controls the default table distribution policy if the `DISTRIBUTED BY` clause is not specified when you create a table. Greenplum Database follows these rules to create a table if a distribution policy is not specified.

- If the Postgres Planner creates the table, and the value of the parameter is `off`, the table distribution policy is determined based on the command.
- If the Postgres Planner creates the table, and the value of the parameter is `on`, the table distribution policy is random.
- If GPORCA creates the table, the table distribution policy is random. The parameter value has no effect.

For more information about setting the default table distribution policy, see *gp_create_table_random_default_distribution*. For information about the Postgres Planner and GPORCA, see *Querying Data* in the *Greenplum Database Administrator Guide*.

Notes

This command is functionally similar to *SELECT INTO*, but it is preferred since it is less likely to be confused with other uses of the `SELECT INTO` syntax. Furthermore, `CREATE TABLE AS` offers a superset of the functionality offered by `SELECT INTO`.

`CREATE TABLE AS` can be used for fast data loading from external table data sources. See *CREATE EXTERNAL TABLE*.

Examples

Create a new table `films_recent` consisting of only recent entries from the table `films`:

```
CREATE TABLE films_recent AS SELECT * FROM films WHERE
date_prod >= '2007-01-01';
```

Create a new temporary table `films_recent`, consisting of only recent entries from the table films, using a prepared statement. The new table will be dropped at commit:

```
PREPARE recentfilms(date) AS SELECT * FROM films WHERE
date_prod > $1;
```

```
CREATE TEMP TABLE films_recent ON COMMIT DROP AS
EXECUTE recentfilms('2007-01-01');
```

Compatibility

`CREATE TABLE AS` conforms to the SQL standard, with the following exceptions:

- The standard requires parentheses around the subquery clause; in Greenplum Database, these parentheses are optional.
- The standard defines a `WITH [NO] DATA` clause; this is not currently implemented by Greenplum Database. The behavior provided by Greenplum Database is equivalent to the standard's `WITH DATA` case. `WITH NO DATA` can be simulated by appending `LIMIT 0` to the query.
- Greenplum Database handles temporary tables differently from the standard; see `CREATE TABLE` for details.
- The `WITH` clause is a Greenplum Database extension; neither storage parameters nor `OIDs` are in the standard. The syntax for creating OID system columns is deprecated and will be removed in a future Greenplum release.
- The Greenplum Database concept of tablespaces is not part of the standard. The `TABLESPACE` clause is an extension.

See Also

CREATE EXTERNAL TABLE, *CREATE EXTERNAL TABLE*, *EXECUTE*, *SELECT*, *SELECT INTO*, *VALUES*

CREATE TABLESPACE

Defines a new tablespace.

Synopsis

```
CREATE TABLESPACE tablespace_name [OWNER username]  LOCATION '/path/to/dir'
   [WITH (contentID_1='/path/to/dir1'[, contentID_2='/path/to/dir2' ... ])]
```

Description

`CREATE TABLESPACE` registers and configures a new tablespace for your Greenplum Database system. The tablespace name must be distinct from the name of any existing tablespace in the system. A tablespace is a Greenplum Database system object (a global object), you can use a tablespace from any database if you have appropriate privileges.

A tablespace allows superusers to define an alternative host file system location where the data files containing database objects (such as tables and indexes) reside.

A user with appropriate privileges can pass a tablespace name to *CREATE DATABASE*, *CREATE TABLE*, or *CREATE INDEX* to have the data files for these objects stored within the specified tablespace.

In Greenplum Database, the file system location must exist on all hosts including the hosts running the master, standby mirror, each primary segment, and each mirror segment.

Parameters

tablespacename

The name of a tablespace to be created. The name cannot begin with pg_ or gp_, as such names are reserved for system tablespaces.

OWNER *username*

The name of the user who will own the tablespace. If omitted, defaults to the user executing the command. Only superusers can create tablespaces, but they can assign ownership of tablespaces to non-superusers.

LOCATION '*/path/to/dir*'

The absolute path to the directory (host system file location) that will be the root directory for the tablespace. When registering a tablepace, the directory should be empty and must be owned by the Greenplum Database system user. The directory must be specified by an absolute path name of no more than 100 characters. (The location is used to create a symlink target in the `pg_tblspc` directory, and symlink targets are truncated to 100 characters when sending to `tar` from utilities such as `pg_basebackup`.)

For each segment instance, you can specify a different directory for the tablespace in the `WITH` clause.

content*ID_i*='*/path/to/dir_i*'

The value *ID_i* is the content ID for the segment instance. */path/to/dir_i* is the absolute path to the host system file location that the segment instance uses as the root directory for the tablespace. You cannot specify the content ID of the master instance (`-1`). You can specify the same directory for multiple segments.

If a segment instance is not listed in the `WITH` clause, Greenplum Database uses the directory specified in the `LOCATION` clause.

When registering a tablepace, the directories should be empty and must be owned by the Greenplum Database system user. Each directory must be specified by an absolute path name of no more than 100 characters.

Notes

Tablespaces are only supported on systems that support symbolic links.

`CREATE TABLESPACE` cannot be executed inside a transaction block.

When creating tablespaces, ensure that file system locations have sufficient I/O speed and available disk space.

`CREATE TABLESPACE` creates symbolic links from the `pg_tblspc` directory in the master and segment instance data directory to the directories specified in the command.

The system catalog table `pg_tablespace` stores tablespace information. This command displays the tablespace OID values, names, and owner.

```
SELECT oid, spcname, spcowner FROM pg_tablespace ;
```

The Greenplum Database built-in function `gp_tablespace_location(`*`tablespace_oid`*`)` displays the tablespace host system file locations for all segment instances. This command lists the segment database IDs and host system file locations for the tablespace with OID `16385`.

```
SELECT * FROM gp_tablespace_location(16385)
```

Examples

Create a new tablespace and specify the file system location for the master and all segment instances:

```
CREATE TABLESPACE mytblspace LOCATION '/gpdbtspc/mytestspace' ;
```

Create a new tablespace and specify a location for segment instances with content ID 0 and 1. For the master and segment instances not listed in the `WITH` clause, the file system location for the tablespace is specified in the `LOCATION` clause.

```
CREATE TABLESPACE mytblspace LOCATION '/gpdbtspc/mytestspace' WITH
 (content0='/temp/mytest', content1='/temp/mytest');
```

The example specifies the same location for the two segment instances. You can a specify different location for each segment.

Compatibility

`CREATE TABLESPACE` is a Greenplum Database extension.

See Also

`CREATE DATABASE`, *`CREATE TABLE`*, *`CREATE INDEX`*, *`DROP TABLESPACE`*, *`ALTER TABLESPACE`*

CREATE TEXT SEARCH CONFIGURATION

Defines a new text search configuration.

Synopsis

```
CREATE TEXT SEARCH CONFIGURATION name (
    PARSER = parser_name |
    COPY = source_config
)
```

Description

`CREATE TEXT SEARCH CONFIGURATION` creates a new text search configuration. A text search configuration specifies a text search parser that can divide a string into tokens, plus dictionaries that can be used to determine which tokens are of interest for searching.

If only the parser is specified, then the new text search configuration initially has no mappings from token types to dictionaries, and therefore will ignore all words. Subsequent `ALTER TEXT SEARCH CONFIGURATION` commands must be used to create mappings to make the configuration useful. Alternatively, an existing text search configuration can be copied.

If a schema name is given then the text search configuration is created in the specified schema. Otherwise it is created in the current schema.

The user who defines a text search configuration becomes its owner.

Refer to *Using Full Text Search* for further information.

Parameters

name

The name of the text search configuration to be created. The name can be schema-qualified.

parser_name

The name of the text search parser to use for this configuration.

source_config

The name of an existing text search configuration to copy.

Notes

The PARSER and COPY options are mutually exclusive, because when an existing configuration is copied, its parser selection is copied too.

Compatibility

There is no CREATE TEXT SEARCH CONFIGURATION statement in the SQL standard.

See Also

ALTER TEXT SEARCH CONFIGURATION, *DROP TEXT SEARCH CONFIGURATION*

CREATE TEXT SEARCH DICTIONARY

Defines a new text search dictionary.

Synopsis

```
CREATE TEXT SEARCH DICTIONARY name (
    TEMPLATE = template
    [, option = value [, ... ]]
)
```

Description

CREATE TEXT SEARCH DICTIONARY creates a new text search dictionary. A text search dictionary specifies a way of recognizing interesting or uninteresting words for searching. A dictionary depends on a text search template, which specifies the functions that actually perform the work. Typically the dictionary provides some options that control the detailed behavior of the template's functions.

If a schema name is given then the text search dictionary is created in the specified schema. Otherwise it is created in the current schema.

The user who defines a text search dictionary becomes its owner.

Refer to *Using Full Text Search* for further information.

Parameters

name

The name of the text search dictionary to be created. The name can be schema-qualified.

template

The name of the text search template that will define the basic behavior of this dictionary.

option

The name of a template-specific option to be set for this dictionary.

value

The value to use for a template-specific option. If the value is not a simple identifier or number, it must be quoted (but you can always quote it, if you wish).

The options can appear in any order.

Examples

The following example command creates a Snowball-based dictionary with a nonstandard list of stop words.

```
CREATE TEXT SEARCH DICTIONARY my_russian (
```

```
    template = snowball,
    language = russian,
    stopwords = myrussian
);
```

Compatibility

There is no `CREATE TEXT SEARCH DICTIONARY` statement in the SQL standard.

See Also

`ALTER TEXT SEARCH DICTIONARY`, *`DROP TEXT SEARCH DICTIONARY`*

CREATE TEXT SEARCH PARSER

Description

Defines a new text search parser.

Synopsis

```
CREATE TEXT SEARCH PARSER name (
    START = start_function ,
    GETTOKEN = gettoken_function ,
    END = end_function ,
    LEXTYPES = lextypes_function
    [, HEADLINE = headline_function ]
)
```

Description

`CREATE TEXT SEARCH PARSER` creates a new text search parser. A text search parser defines a method for splitting a text string into tokens and assigning types (categories) to the tokens. A parser is not particularly useful by itself, but must be bound into a text search configuration along with some text search dictionaries to be used for searching.

If a schema name is given then the text search parser is created in the specified schema. Otherwise it is created in the current schema.

You must be a superuser to use `CREATE TEXT SEARCH PARSER`. (This restriction is made because an erroneous text search parser definition could confuse or even crash the server.)

Refer to *Using Full Text Search* for further information.

Parameters

`name`

The name of the text search parser to be created. The name can be schema-qualified.

`start_function`

The name of the start function for the parser.

`gettoken_function`

The name of the get-next-token function for the parser.

`end_function`

The name of the end function for the parser.

`lextypes_function`

The name of the lextypes function for the parser (a function that returns information about the set of token types it produces).

headline_function

The name of the headline function for the parser (a function that summarizes a set of tokens).

The function names can be schema-qualified if necessary. Argument types are not given, since the argument list for each type of function is predetermined. All except the headline function are required.

The arguments can appear in any order, not only the one shown above.

Compatibility

There is no `CREATE TEXT SEARCH PARSER` statement in the SQL standard.

See Also

ALTER TEXT SEARCH PARSER, *DROP TEXT SEARCH PARSER*

CREATE TEXT SEARCH TEMPLATE

Description

Defines a new text search template.

Synopsis

```
CREATE TEXT SEARCH TEMPLATE name (
    [ INIT = init_function , ]
    LEXIZE = lexize_function
)
```

Description

`CREATE TEXT SEARCH TEMPLATE` creates a new text search template. Text search templates define the functions that implement text search dictionaries. A template is not useful by itself, but must be instantiated as a dictionary to be used. The dictionary typically specifies parameters to be given to the template functions.

If a schema name is given then the text search template is created in the specified schema. Otherwise it is created in the current schema.

You must be a superuser to use `CREATE TEXT SEARCH TEMPLATE`. This restriction is made because an erroneous text search template definition could confuse or even crash the server. The reason for separating templates from dictionaries is that a template encapsulates the "unsafe" aspects of defining a dictionary. The parameters that can be set when defining a dictionary are safe for unprivileged users to set, and so creating a dictionary need not be a privileged operation.

Refer to *Using Full Text Search* for further information.

Parameters

name

The name of the text search template to be created. The name can be schema-qualified.

init_function

The name of the init function for the template.

lexize_function

The name of the lexize function for the template.

The function names can be schema-qualified if necessary. Argument types are not given, since the argument list for each type of function is predetermined. The lexize function is required, but the init function is optional.

The arguments can appear in any order, not only the order shown above.

Compatibility

There is no `CREATE TEXT SEARCH TEMPLATE` statement in the SQL standard.

See Also

CREATE TEXT SEARCH TEMPLATE, *DROP TEXT SEARCH TEMPLATE*

CREATE TYPE

Defines a new data type.

Synopsis

```
CREATE TYPE name AS
    ( attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ]
    [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ]
)

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [, RECEIVE = receive_function]
    [, SEND = send_function]
    [, TYPMOD_IN = type_modifier_input_function ]
    [, TYPMOD_OUT = type_modifier_output_function ]
    [, INTERNALLENGTH = {internallength | VARIABLE}]
    [, PASSEDBYVALUE]
    [, ALIGNMENT = alignment]
    [, STORAGE = storage]
    [, LIKE = like_type
    [, CATEGORY = category]
    [, PREFERRED = preferred]
    [, DEFAULT = default]
    [, ELEMENT = element]
    [, DELIMITER = delimiter]
    [, COLLATABLE = collatable]
    [, COMPRESSTYPE = compression_type]
    [, COMPRESSLEVEL = compression_level]
    [, BLOCKSIZE = blocksize] )

CREATE TYPE name
```

Description

CREATE TYPE registers a new data type for use in the current database. The user who defines a type becomes its owner.

If a schema name is given then the type is created in the specified schema. Otherwise it is created in the current schema. The type name must be distinct from the name of any existing type or domain in the same schema. The type name must also be distinct from the name of any existing table in the same schema.

There are five forms of CREATE TYPE, as shown in the syntax synopsis above. They respectively create a *composite type*, an *enum type*, a *range type*, a *base type*, or a *shell type*. The first four of these are discussed in turn below. A shell type is simply a placeholder for a type to be defined later; it is created by issuing CREATE TYPE with no parameters except for the type name. Shell types are needed as forward references when creating range types and base types, as discussed in those sections.

Composite Types

The first form of CREATE TYPE creates a composite type. The composite type is specified by a list of attribute names and data types. An attribute's collation can be specified too, if its data type is collatable. A composite type is essentially the same as the row type of a table, but using CREATE TYPE avoids the need to create an actual table when all that is wanted is to define a type. A stand-alone composite type is useful, for example, as the argument or return type of a function.

To be able to create a composite type, you must have USAGE privilege on all attribute types.

Enumerated Types

The second form of CREATE TYPE creates an enumerated (ENUM) type, as described in *Enumerated Types* in the PostgreSQL documentation. ENUM types take a list of quoted labels, each of which must be less than NAMEDATALEN bytes long (64 in a standard build).

It is possible to create an enumerated type with zero labels, but such a type cannot be used to hold values before at least one label is added using *ALTER TYPE*.

Range Types

The third form of CREATE TYPE creates a new range type, as described in *Range Types*.

The range type's *subtype* can be any type with an associated b-tree operator class (to determine the ordering of values for the range type). Normally the subtype's default b-tree operator class is used to determine ordering; to use a non-default operator class, specify its name with *subtype_opclass*. If the subtype is collatable, and you want to use a non-default collation in the range's ordering, specify the desired collation with the *collation* option.

The optional *canonical* function must take one argument of the range type being defined, and return a value of the same type. This is used to convert range values to a canonical form, when applicable. See Section *Defining New Range Types* for more information. Creating a *canonical* function is a bit tricky, since it must be defined before the range type can be declared. To do this, you must first create a shell type, which is a placeholder type that has no properties except a name and an owner. This is done by issuing the command CREATE TYPE *name*, with no additional parameters. Then the function can be declared using the shell type as argument and result, and finally the range type can be declared using the same name. This automatically replaces the shell type entry with a valid range type.

The optional *subtype_diff* function must take two values of the *subtype* type as argument, and return a double precision value representing the difference between the two given values. While this is optional, providing it allows much greater efficiency of GiST indexes on columns of the range type. See *Defining New Range Types* for more information.

Base Types

The fourth form of CREATE TYPE creates a new base type (scalar type). You must be a superuser to create a new base type. The parameters may appear in any order, not only that shown in the syntax, and most are optional. You must register two or more functions (using CREATE FUNCTION) before defining the type. The support functions *input_function* and *output_function* are required, while the functions

receive_function, *send_function*, *type_modifier_input_function*, *type_modifier_output_function*, and *analyze_function* are optional. Generally these functions have to be coded in C or another low-level language. In Greenplum Database, any function used to implement a data type must be defined as `IMMUTABLE`.

The *input_function* converts the type's external textual representation to the internal representation used by the operators and functions defined for the type. *output_function* performs the reverse transformation. The input function may be declared as taking one argument of type `cstring`, or as taking three arguments of types `cstring`, `oid`, `integer`. The first argument is the input text as a C string, the second argument is the type's own OID (except for array types, which instead receive their element type's OID), and the third is the `typmod` of the destination column, if known (`-1` will be passed if not). The input function must return a value of the data type itself. Usually, an input function should be declared `STRICT`; if it is not, it will be called with a `NULL` first parameter when reading a `NULL` input value. The function must still return `NULL` in this case, unless it raises an error. (This case is mainly meant to support domain input functions, which may need to reject `NULL` inputs.) The output function must be declared as taking one argument of the new data type. The output function must return type `cstring`. Output functions are not invoked for `NULL` values.

The optional *receive_function* converts the type's external binary representation to the internal representation. If this function is not supplied, the type cannot participate in binary input. The binary representation should be chosen to be cheap to convert to internal form, while being reasonably portable. (For example, the standard integer data types use network byte order as the external binary representation, while the internal representation is in the machine's native byte order.) The receive function should perform adequate checking to ensure that the value is valid. The receive function may be declared as taking one argument of type `internal`, or as taking three arguments of types `internal`, `oid`, `integer`. The first argument is a pointer to a `StringInfo` buffer holding the received byte string; the optional arguments are the same as for the text input function. The receive function must return a value of the data type itself. Usually, a receive function should be declared `STRICT`; if it is not, it will be called with a `NULL` first parameter when reading a NULL input value. The function must still return `NULL` in this case, unless it raises an error. (This case is mainly meant to support domain receive functions, which may need to reject `NULL` inputs.) Similarly, the optional *send_function* converts from the internal representation to the external binary representation. If this function is not supplied, the type cannot participate in binary output. The send function must be declared as taking one argument of the new data type. The send function must return type `bytea`. Send functions are not invoked for `NULL` values.

The optional *type_modifier_input_function* and *type_modifier_output_function* are required if the type supports modifiers. Modifiers are optional constraints attached to a type declaration, such as `char(5)` or `numeric(30,2)`. While Greenplum Database allows user-defined types to take one or more simple constants or identifiers as modifiers, this information must fit into a single non-negative integer value for storage in the system catalogs. Greenplum Database passes the declared modifier(s) to the *type_modifier_input_function* in the form of a `cstring` array. The modifier input function must check the values for validity, throwing an error if they are incorrect. If the values are correct, the modifier input function returns a single non-negative integer value that Greenplum Database stores as the column `typmod`. Type modifiers are rejected if the type was not defined with a *type_modifier_input_function*. The *type_modifier_output_function* converts the internal integer `typmod` value back to the correct form for user display. The modifier output function must return a `cstring` value that is the exact string to append to the type name. For example, `numeric`'s function might return `(30,2)`. The *type_modifier_output_function* is optional. When not specified, the default display format is the stored `typmod` integer value enclosed in parentheses.

You should at this point be wondering how the input and output functions can be declared to have results or arguments of the new type, when they have to be created before the new type can be created. The answer is that the type should first be defined as a shell type, which is a placeholder type that has no properties except a name and an owner. This is done by issuing the command `CREATE TYPE` *name*, with no additional parameters. Then the I/O functions can be defined referencing the shell type. Finally, `CREATE TYPE` with a full definition replaces the shell entry with a complete, valid type definition, after which the new type can be used normally.

The *like_type* parameter provides an alternative method for specifying the basic representation properties of a data type: copy them from some existing type. The values `internallength`, `passedbyvalue`, `alignment`, and `storage` are copied from the named type. (It is possible, though usually undesirable, to override some of these values by specifying them along with the `LIKE` clause.) Specifying representation this way is especially useful when the low-level implementation of the new type "piggybacks" on an existing type in some fashion.

While the details of the new type's internal representation are only known to the I/O functions and other functions you create to work with the type, there are several properties of the internal representation that must be declared to Greenplum Database. Foremost of these is *internallength*. Base data types can be fixed-length, in which case *internallength* is a positive integer, or variable length, indicated by setting *internallength* to `VARIABLE`. (Internally, this is represented by setting `typlen` to `-1`.) The internal representation of all variable-length types must start with a 4-byte integer giving the total length of this value of the type.

The optional flag `PASSEDBYVALUE` indicates that values of this data type are passed by value, rather than by reference. You may not pass by value types whose internal representation is larger than the size of the `Datum` type (4 bytes on most machines, 8 bytes on a few).

The *alignment* parameter specifies the storage alignment required for the data type. The allowed values equate to alignment on 1, 2, 4, or 8 byte boundaries. Note that variable-length types must have an alignment of at least 4, since they necessarily contain an `int4` as their first component.

The *storage* parameter allows selection of storage strategies for variable-length data types. (Only `plain` is allowed for fixed-length types.) `plain` specifies that data of the type will always be stored in-line and not compressed. `extended` specifies that the system will first try to compress a long data value, and will move the value out of the main table row if it's still too long. `external` allows the value to be moved out of the main table, but the system will not try to compress it. `main` allows compression, but discourages moving the value out of the main table. (Data items with this storage strategy may still be moved out of the main table if there is no other way to make a row fit, but they will be kept in the main table preferentially over `extended` and `external` items.)

A default value may be specified, in case a user wants columns of the data type to default to something other than the null value. Specify the default with the `DEFAULT` key word. (Such a default may be overridden by an explicit `DEFAULT` clause attached to a particular column.)

To indicate that a type is an array, specify the type of the array elements using the `ELEMENT` key word. For example, to define an array of 4-byte integers (int4), specify `ELEMENT = int4`. More details about array types appear below.

The *category* and *preferred* parameters can be used to help control which implicit cast Greenplum Database applies in ambiguous situations. Each data type belongs to a category named by a single ASCII character, and each type is either "preferred" or not within its category. The parser will prefer casting to preferred types (but only from other types within the same category) when this rule helps resolve overloaded functions or operators. For types that have no implicit casts to or from any other types, it is sufficient to retain the default settings. However, for a group of related types that have implicit casts, it is often helpful to mark them all as belonging to a category and select one or two of the "most general" types as being preferred within the category. The *category* parameter is especially useful when you add a user-defined type to an existing built-in category, such as the numeric or string types. It is also possible to create new entirely-user-defined type categories. Select any ASCII character other than an upper-case letter to name such a category.

To indicate the delimiter to be used between values in the external representation of arrays of this type, `delimiter` can be set to a specific character. The default delimiter is the comma (,). Note that the delimiter is associated with the array element type, not the array type itself.

If the optional Boolean parameter *collatable* is true, column definitions and expressions of the type may carry collation information through use of the `COLLATE` clause. It is up to the implementations of the functions operating on the type to actually make use of the collation information; this does not happen automatically merely by marking the type collatable.

Array Types

Whenever a user-defined type is created, Greenplum Database automatically creates an associated array type, whose name consists of the element type's name prepended with an underscore, and truncated if necessary to keep it less than `NAMEDATALEN` bytes long. (If the name so generated collides with an existing type name, the process is repeated until a non-colliding name is found.) This implicitly-created array type is variable length and uses the built-in input and output functions `array_in` and `array_out`. The array type tracks any changes in its element type's owner or schema, and is dropped if the element type is.

You might reasonably ask why there is an `ELEMENT` option, if the system makes the correct array type automatically. The only case where it's useful to use `ELEMENT` is when you are making a fixed-length type that happens to be internally an array of a number of identical things, and you want to allow these things to be accessed directly by subscripting, in addition to whatever operations you plan to provide for the type as a whole. For example, type `point` is represented as just two floating-point numbers, each can be accessed using `point[0]` and `point[1]`. Note that this facility only works for fixed-length types whose internal form is exactly a sequence of identical fixed-length fields. A subscriptable variable-length type must have the generalized internal representation used by `array_in` and `array_out`. For historical reasons (i.e., this is clearly wrong but it's far too late to change it), subscripting of fixed-length array types starts from zero, rather than from one as for variable-length arrays.

Parameters

name

The name (optionally schema-qualified) of a type to be created.

attribute_name

The name of an attribute (column) for the composite type.

data_type

The name of an existing data type to become a column of the composite type.

collation

The name of an existing collation to be associated with a column of a composite type, or with a range type.

label

A string literal representing the textual label associated with one value of an enum type.

subtype

The name of the element type that the range type will represent ranges of.

subtype_operator_class

The name of a b-tree operator class for the subtype.

canonical_function

The name of the canonicalization function for the range type.

subtype_diff_function

The name of a difference function for the subtype.

input_function

The name of a function that converts data from the type's external textual form to its internal form.

output_function

The name of a function that converts data from the type's internal form to its external textual form.

receive_function

The name of a function that converts data from the type's external binary form to its internal form.

send_function

The name of a function that converts data from the type's internal form to its external binary form.

type_modifier_input_function

The name of a function that converts an array of modifier(s) for the type to internal form.

type_modifier_output_function

The name of a function that converts the internal form of the type's modifier(s) to external textual form.

internallength

A numeric constant that specifies the length in bytes of the new type's internal representation. The default assumption is that it is variable-length.

alignment

The storage alignment requirement of the data type. Must be one of `char`, `int2`, `int4`, or `double`. The default is `int4`.

storage

The storage strategy for the data type. Must be one of `plain`, `external`, `extended`, or `main`. The default is `plain`.

like_type

The name of an existing data type that the new type will have the same representation as. The values *internallength*, *passedbyvalue*, *alignment*, and *storage*, are copied from that type, unless overridden by explicit specification elsewhere in this `CREATE TYPE` command.

category

The category code (a single ASCII character) for this type. The default is 'U', signifying a user-defined type. You can find the other standard category codes in *pg_type Category Codes*. You may also assign unused ASCII characters to custom categories that you create.

preferred

`true` if this type is a preferred type within its type category, else `false`. The default value is `false`. Be careful when you create a new preferred type within an existing type category; this could cause surprising behaviour changes.

default

The default value for the data type. If this is omitted, the default is null.

element

The type being created is an array; this specifies the type of the array elements.

delimiter

The delimiter character to be used between values in arrays made of this type.

collatable

True if this type's operations can use collation information. The default is false.

compression_type

Set to `ZLIB` (the default), `ZSTD`, `RLE_TYPE`, or `QUICKLZ`[1] to specify the type of compression used in columns of this type.

> **Note:** [1]QuickLZ compression is available only in the commercial release of Pivotal Greenplum Database.

compression_level

For Zstd compression, set to an integer value from 1 (fastest compression) to 19 (highest compression ratio). For zlib compression, the valid range is from 1 to 9. The QuickLZ compression level can only be set to 1. For `RLE_TYPE`, the compression level can be set to an integer value from 1 (fastest compression) to 4 (highest compression ratio). The default compression level is 1.

blocksize

Set to the size, in bytes, for each block in the column. The `BLOCKSIZE` must be between 8192 and 2097152 bytes, and be a multiple of 8192. The default block size is 32768.

Notes

User-defined type names cannot begin with the underscore character (_) and can only be 62 characters long (or in general `NAMEDATALEN - 2`, rather than the `NAMEDATALEN - 1` characters allowed for other names). Type names beginning with underscore are reserved for internally-created array type names.

Greenplum Database does not support adding storage options for row or composite types.

Storage options defined at the table- and column- level override the default storage options defined for a scalar type.

Because there are no restrictions on use of a data type once it's been created, creating a base type or range type is tantamount to granting public execute permission on the functions mentioned in the type definition. (The creator of the type is therefore required to own these functions.) This is usually not an issue for the sorts of functions that are useful in a type definition. But you might want to think twice before designing a type in a way that would require 'secret' information to be used while converting it to or from external form.

Examples

This example creates a composite type and uses it in a function definition:

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

This example creates the enumerated type `mood` and uses it in a table definition.

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

This example creates a range type:

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff =
 float8mi);
```

This example creates the base data type box and then uses the type in a table definition:

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS
... ;

CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS
... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

If the internal structure of box were an array of four `float4` elements, we might instead use:

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

which would allow a box value's component numbers to be accessed by subscripting. Otherwise the type behaves the same as before.

This example creates a large object type and uses it in a table definition:

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);

CREATE TABLE big_objs (
    id integer,
    obj bigobj
    );
```

Compatibility

The first form of the CREATE TYPE command, which creates a composite type, conforms to the SQL standard. The other forms are Greenplum Database extensions. The CREATE TYPE statement in the SQL standard also defines other forms that are not implemented in Greenplum Database.

The ability to create a composite type with zero attributes is a Greenplum Database-specific deviation from the standard (analogous to the same case in CREATE TABLE).

See Also

ALTER TYPE, *CREATE DOMAIN*, *CREATE FUNCTION*, *DROP TYPE*

CREATE USER

Defines a new database role with the LOGIN privilege by default.

Synopsis

```
CREATE USER name [[WITH] option [ ... ]]
```

where *option* can be:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | CREATEEXTTABLE | NOCREATEEXTTABLE
      [ ( attribute='value'[, ...] ) ]
           where attributes and value are:
           type='readable'|'writable'
           protocol='gpfdist'|'http'
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
    | RESOURCE QUEUE queue_name
    | RESOURCE GROUP group_name
    | [ DENY deny_point ]
    | [ DENY BETWEEN deny_point AND deny_point]
```

Description

`CREATE USER` is an alias for *CREATE ROLE*.

The only difference between `CREATE ROLE` and `CREATE USER` is that `LOGIN` is assumed by default with `CREATE USER`, whereas `NOLOGIN` is assumed by default with `CREATE ROLE`.

Compatibility

There is no `CREATE USER` statement in the SQL standard.

See Also

CREATE ROLE

CREATE USER MAPPING

Defines a new mapping of a user to a foreign server.

Synopsis

```
CREATE USER MAPPING FOR { username | USER | CURRENT_USER | PUBLIC }
    SERVER servername
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

Description

`CREATE USER MAPPING` defines a mapping of a user to a foreign server. You must be the owner of the server to define user mappings for it.

Parameters

username

The name of an existing user that is mapped to the foreign server. `CURRENT_USER` and `USER` match the name of the current user. `PUBLIC` is used to match all present and future user names in the system.

servername

The name of an existing server for which Greenplum Database is to create the user mapping.

`OPTIONS ( `*`option`*` '`*`value`*`' [, ... ] )`

The options for the new user mapping. The options typically define the actual user name and password of the mapping. Option names must be unique. The option names and values are specific to the server's foreign-data wrapper.

Examples

Create a user mapping for user `bob`, server `foo`:

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password
 'secret');
```

Compatibility

`CREATE USER MAPPING` conforms to ISO/IEC 9075-9 (SQL/MED).

See Also

`ALTER USER MAPPING`, *`DROP USER MAPPING`*, *`CREATE FOREIGN DATA WRAPPER`*, *`CREATE SERVER`*

CREATE VIEW

Defines a new view.

Synopsis

```
CREATE [OR REPLACE] [TEMP | TEMPORARY] [RECURSIVE] VIEW name [ ( column_name
 [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Description

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

`CREATE OR REPLACE VIEW` is similar, but if a view of the same name already exists, it is replaced. The new query must generate the same columns that were generated by the existing view query (that is, the same column names in the same order, and with the same data types), but it may add additional columns to the end of the list. The calculations giving rise to the output columns may be completely different.

If a schema name is given then the view is created in the specified schema. Otherwise it is created in the current schema. Temporary views exist in a special schema, so a schema name may not be given when creating a temporary view. The name of the view must be distinct from the name of any other view, table, sequence, index or foreign table in the same schema.

Parameters

TEMPORARY | TEMP

If specified, the view is created as a temporary view. Temporary views are automatically dropped at the end of the current session. Existing permanent relations with the same name are not visible to the current session while the temporary view exists, unless they are referenced with schema-qualified names. If any of the tables referenced by the view are temporary, the view is created as a temporary view (whether TEMPORARY is specified or not).

RECURSIVE

Creates a recursive view. The syntax

```
CREATE RECURSIVE VIEW [ schema . ] view_name (column_names) AS
 SELECT ...;
```

is equivalent to

```
CREATE VIEW [ schema . ] view_name AS WITH RECURSIVE view_name
 (column_names) AS (SELECT ...) SELECT column_names
 FROM view_name;
```

A view column name list must be specified for a recursive view.

name

The name (optionally schema-qualified) of a view to be created.

column_name

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

WITH (view_option_name [= view_option_value] [, ...])

This clause specifies optional parameters for a view; the following parameters are supported:

check_option (string)

This parameter may be either `local` or `cascaded`, and is equivalent to specifying `WITH [ CASCADED | LOCAL ] CHECK OPTION` (see below). This option can be changed on existing views using *ALTER VIEW*.

security_barrier (boolean)

This should be used if the view is intended to provide row-level security.

query

A *SELECT* or *VALUES* command which will provide the columns and rows of the view.

Notes

Views in Greenplum Database are read only. The system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rewrite rules on the view into appropriate actions on other tables. For more information see `CREATE RULE`.

Be careful that the names and data types of the view's columns will be assigned the way you want. For example:

```
CREATE VIEW vista AS SELECT 'Hello World';
```

is bad form in two ways: the column name defaults to `?column?`, and the column data type defaults to `unknown`. If you want a string literal in a view's result, use something like:

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

Access to tables referenced in the view is determined by permissions of the view owner not the current user (even if the current user is a superuser). This can be confusing in the case of superusers, since superusers typically have access to all objects. In the case of a view, even superusers must be explicitly granted access to tables referenced in the view if they are not the owner of the view.

However, functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore the user of a view must have permissions to call any functions used by the view.

If you create a view with an `ORDER BY` clause, the `ORDER BY` clause is ignored when you do a `SELECT` from the view.

When `CREATE OR REPLACE VIEW` is used on an existing view, only the view's defining `SELECT` rule is changed. Other view properties, including ownership, permissions, and non-`SELECT` rules, remain unchanged. You must own the view to replace it (this includes being a member of the owning role).

Examples

Create a view consisting of all comedy films:

```
CREATE VIEW comedies AS SELECT * FROM films
WHERE kind = 'comedy';
```

This will create a view containing the columns that are in the `film` table at the time of view creation. Though * was used to create the view, columns added later to the table will not be part of the view.

Create a view that gets the top ten ranked baby names:

```
CREATE VIEW topten AS SELECT name, rank, gender, year FROM
names, rank WHERE rank < '11' AND names.id=rank.id;
```

Create a recursive view consisting of the numbers from 1 to 100:

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
    VALUES (1)
UNION ALL
    SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

Notice that although the recursive view's name is schema-qualified in this `CREATE VIEW` command, its internal self-reference is not schema-qualified. This is because the implicitly-created CTE's name cannot be schema-qualified.

Compatibility

The SQL standard specifies some additional capabilities for the `CREATE VIEW` statement that are not in Greenplum Database. The optional clauses for the full SQL command in the standard are:

- **CHECK OPTION** — This option has to do with updatable views. All `INSERT` and `UPDATE` commands on the view will be checked to ensure data satisfy the view-defining condition (that is, the new data would be visible through the view). If they do not, the update will be rejected.
- **LOCAL** — Check for integrity on this view.
- **CASCADED** — Check for integrity on this view and on any dependent view. `CASCADED` is assumed if neither `CASCADED` nor `LOCAL` is specified.

`CREATE OR REPLACE VIEW` is a Greenplum Database language extension. So is the concept of a temporary view.

See Also

SELECT, *DROP VIEW*, *CREATE MATERIALIZED VIEW*

DEALLOCATE

Deallocates a prepared statement.

Synopsis

```
DEALLOCATE [PREPARE] name
```

Description

`DEALLOCATE` is used to deallocate a previously prepared SQL statement. If you do not explicitly deallocate a prepared statement, it is deallocated when the session ends.

For more information on prepared statements, see *PREPARE*.

Parameters

PREPARE

Optional key word which is ignored.

name

The name of the prepared statement to deallocate.

Examples

Deallocated the previously prepared statement named `insert_names`:

```
DEALLOCATE insert_names;
```

Compatibility

The SQL standard includes a `DEALLOCATE` statement, but it is only for use in embedded SQL.

See Also

EXECUTE, *PREPARE*

DECLARE

Defines a cursor.

Synopsis

```
DECLARE name [BINARY] [INSENSITIVE] [NO SCROLL] CURSOR
```

```
    [{WITH | WITHOUT} HOLD]
    FOR query [FOR READ ONLY]
```

Description

DECLARE allows a user to create cursors, which can be used to retrieve a small number of rows at a time out of a larger query. Cursors can return data either in text or in binary format using *FETCH*.

> **Note:** This page describes usage of cursors at the SQL command level. If you are trying to use cursors inside a PL/pgSQL function, the rules are different, see *PL/Sql*.

Normal cursors return data in text format, the same as a SELECT would produce. Since data is stored natively in binary format, the system must do a conversion to produce the text format. Once the information comes back in text form, the client application may need to convert it to a binary format to manipulate it. In addition, data in the text format is often larger in size than in the binary format. Binary cursors return the data in a binary representation that may be more easily manipulated. Nevertheless, if you intend to display the data as text anyway, retrieving it in text form will save you some effort on the client side.

As an example, if a query returns a value of one from an integer column, you would get a string of 1 with a default cursor whereas with a binary cursor you would get a 4-byte field containing the internal representation of the value (in big-endian byte order).

Binary cursors should be used carefully. Many applications, including psql, are not prepared to handle binary cursors and expect data to come back in the text format.

> **Note:**
>
> When the client application uses the 'extended query' protocol to issue a FETCH command, the Bind protocol message specifies whether data is to be retrieved in text or binary format. This choice overrides the way that the cursor is defined. The concept of a binary cursor as such is thus obsolete when using extended query protocol — any cursor can be treated as either text or binary.

A cursor can be specified in the WHERE CURRENT OF clause of the *UPDATE* or *DELETE* statement to update or delete table data. The UPDATE or DELETE statement can only be executed on the server, for example in an interactive psql session or a script. Language extensions such as PL/pgSQL do not have support for updatable cursors.

Parameters

name

The name of the cursor to be created.

BINARY

Causes the cursor to return data in binary rather than in text format.

INSENSITIVE

Indicates that data retrieved from the cursor should be unaffected by updates to the tables underlying the cursor while the cursor exists. In Greenplum Database, all cursors are insensitive. This key word currently has no effect and is present for compatibility with the SQL standard.

NO SCROLL

A cursor cannot be used to retrieve rows in a nonsequential fashion. This is the default behavior in Greenplum Database, since scrollable cursors (SCROLL) are not supported.

WITH HOLD
WITHOUT HOLD

WITH HOLD specifies that the cursor may continue to be used after the transaction that created it successfully commits. WITHOUT HOLD specifies that the cursor cannot be used outside of the transaction that created it. WITHOUT HOLD is the default.

`WITH HOLD` cannot not be specified when the `query` includes a `FOR UPDATE` or `FOR SHARE` clause.

query

A *SELECT* or *VALUES* command which will provide the rows to be returned by the cursor.

If the cursor is used in the `WHERE CURRENT OF` clause of the *UPDATE* or *DELETE* command, the `SELECT` command must satisfy the following conditions:

- Cannot reference a view or external table.
- References only one table.

  The table must be updatable. For example, the following are not updatable: table functions, set-returning functions, append-only tables, columnar tables.
- Cannot contain any of the following:
  - A grouping clause
  - A set operation such as `UNION ALL` or `UNION DISTINCT`
  - A sorting clause
  - A windowing clause
  - A join or a self-join

  Specifying the `FOR UPDATE` clause in the `SELECT` command prevents other sessions from changing the rows between the time they are fetched and the time they are updated. Without the `FOR UPDATE` clause, a subsequent use of the `UPDATE` or `DELETE` command with the `WHERE CURRENT OF` clause has no effect if the row was changed since the cursor was created.

  > **Note:** Specifying the `FOR UPDATE` clause in the `SELECT` command locks the entire table, not just the selected rows.

`FOR READ ONLY`

`FOR READ ONLY` indicates that the cursor is used in a read-only mode.

Notes

Unless `WITH HOLD` is specified, the cursor created by this command can only be used within the current transaction. Thus, `DECLARE` without `WITH HOLD` is useless outside a transaction block: the cursor would survive only to the completion of the statement. Therefore Greenplum Database reports an error if this command is used outside a transaction block. Use `BEGIN` and `COMMIT` (or `ROLLBACK`) to define a transaction block.

If `WITH HOLD` is specified and the transaction that created the cursor successfully commits, the cursor can continue to be accessed by subsequent transactions in the same session. (But if the creating transaction is aborted, the cursor is removed.) A cursor created with `WITH HOLD` is closed when an explicit `CLOSE` command is issued on it, or the session ends. In the current implementation, the rows represented by a held cursor are copied into a temporary file or memory area so that they remain available for subsequent transactions.

If you create a cursor with the `DECLARE` command in a transaction, you cannot use the `SET` command in the transaction until you close the cursor with the `CLOSE` command.

Scrollable cursors are not currently supported in Greenplum Database. You can only use `FETCH` to move the cursor position forward, not backwards.

`DECLARE...FOR UPDATE` is not supported with append-optimized tables.

You can see all available cursors by querying the `pg_cursors` system view.

Examples

Declare a cursor:

```
DECLARE mycursor CURSOR FOR SELECT * FROM mytable;
```

Compatibility

SQL standard allows cursors only in embedded SQL and in modules. Greenplum Database permits cursors to be used interactively.

Greenplum Database does not implement an `OPEN` statement for cursors. A cursor is considered to be open when it is declared.

The SQL standard allows cursors to move both forward and backward. All Greenplum Database cursors are forward moving only (not scrollable).

Binary cursors are a Greenplum Database extension.

See Also

CLOSE, *DELETE*, *FETCH*, *MOVE*, *SELECT*, *UPDATE*

DELETE

Deletes rows from a table.

Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ONLY] table [[AS] alias]
      [USING usinglist]
      [WHERE condition | WHERE CURRENT OF cursor_name]
      [RETURNING * | output_expression [[AS] output_name] [, …]]
```

Description

`DELETE` deletes rows that satisfy the `WHERE` clause from the specified table. If the `WHERE` clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

By default, `DELETE` will delete rows in the specified table and all its child tables. If you wish to delete only from the specific table mentioned, you must use the `ONLY` clause.

There are two ways to delete rows in a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the `USING` clause. Which technique is more appropriate depends on the specific circumstances.

If the `WHERE CURRENT OF` clause is specified, the row that is deleted is the one most recently fetched from the specified cursor.

The `WHERE CURRENT OF` clause is not supported with replicated tables.

The optional `RETURNING` clause causes `DELETE` to compute and return value(s) based on each row actually deleted. Any expression using the table's columns, and/or columns of other tables mentioned in `USING`, can be computed. The syntax of the `RETURNING` list is identical to that of the output list of `SELECT`.

> **Note:** The `RETURNING` clause is not supported when deleting from append-optimized tables.

You must have the `DELETE` privilege on the table to delete from it.

Note: As the default, Greenplum Database acquires an `EXCLUSIVE` lock on tables for `DELETE` operations on heap tables. When the Global Deadlock Detector is enabled, the lock mode for `DELETE` operations on heap tables is `ROW EXCLUSIVE`. See *Global Deadlock Detector*.

Outputs

On successful completion, a `DELETE` command returns a command tag of the form

```
DELETE count
```

The *count* is the number of rows deleted. If *count* is 0, no rows were deleted by the query (this is not considered an error).

If the `DELETE` command contains a `RETURNING` clause, the result will be similar to that of a `SELECT` statement containing the columns and values defined in the `RETURNING` list, computed over the row(s) deleted by the command.

Parameters

with_query

The `WITH` clause allows you to specify one or more subqueries that can be referenced by name in the `DELETE` query.

For a `DELETE` command that includes a `WITH` clause, the clause can only contain `SELECT` statements, the `WITH` clause cannot contain a data-modifying command (`INSERT`, `UPDATE`, or `DELETE`).

See *WITH Queries (Common Table Expressions)* and *SELECT* for details.

ONLY

If specified, delete rows from the named table only. When not specified, any tables inheriting from the named table are also processed.

table

The name (optionally schema-qualified) of an existing table.

alias

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given `DELETE FROM foo AS f`, the remainder of the `DELETE` statement must refer to this table as `f` not `foo`.

usinglist

A list of table expressions, allowing columns from other tables to appear in the `WHERE` condition. This is similar to the list of tables that can be specified in the `FROM` Clause of a *SELECT* statement; for example, an alias for the table name can be specified. Do not repeat the target table in the `usinglist`, unless you wish to set up a self-join.

condition

An expression returning a value of type `boolean`, which determines the rows that are to be deleted.

cursor_name

The name of the cursor to use in a `WHERE CURRENT OF` condition. The row to be deleted is the one most recently fetched from this cursor. The cursor must be a simple non-grouping query on the `DELETE` target table.

`WHERE CURRENT OF` cannot be specified together with a Boolean condition.

The `DELETE...WHERE CURRENT OF` cursor statement can only be executed on the server, for example in an interactive psql session or a script. Language extensions such as PL/pgSQL do not have support for updatable cursors.

See *DECLARE* for more information about creating cursors.

output_expression

An expression to be computed and returned by the DELETE command after each row is deleted. The expression can use any column names of the table or table(s) listed in USING. Write * to return all columns.

output_name

A name to use for a returned column.

Notes

Greenplum Database lets you reference columns of other tables in the WHERE condition by specifying the other tables in the USING clause. For example, to the name Hannah from the rank table, one might do:

```
DELETE FROM rank USING names WHERE names.id = rank.id AND
name = 'Hannah';
```

What is essentially happening here is a join between rank and names, with all successfully joined rows being marked for deletion. This syntax is not standard. However, this join style is usually easier to write and faster to execute than a more standard sub-select style, such as:

```
DELETE FROM rank WHERE id IN (SELECT id FROM names WHERE name
= 'Hannah');
```

When using DELETE to remove all the rows of a table (for example: DELETE * FROM *table*;), Greenplum Database adds an implicit TRUNCATE command (when user permissions allow). The added TRUNCATE command frees the disk space occupied by the deleted rows without requiring a VACUUM of the table. This improves scan performance of subsequent queries, and benefits ELT workloads that frequently insert and delete from temporary tables.

Execution of UPDATE and DELETE commands directly on a specific partition (child table) of a partitioned table is not supported. Instead, these commands must be executed on the root partitioned table, the table created with the CREATE TABLE command.

For a partitioned table, all the child tables are locked during the DELETE operation when the Global Deadlock Detector is not enabled (the default). Only some of the leaf child tables are locked when the Global Deadlock Detector is enabled. For information about the Global Deadlock Detector, see *Global Deadlock Detector*.

Examples

Delete all films but musicals:

```
DELETE FROM films WHERE kind <> 'Musical';
```

Clear the table films:

```
DELETE FROM films;
```

Delete completed tasks, returning full details of the deleted rows:

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

Delete using a join:

```
DELETE FROM rank USING names WHERE names.id = rank.id AND
name = 'Hannah';
```

Compatibility

This command conforms to the SQL standard, except that the `USING` and `RETURNING` clauses are Greenplum Database extensions, as is the ability to use `WITH` with `DELETE`.

See Also

DECLARE, *TRUNCATE*

DISCARD

Discards the session state.

Synopsis

```
DISCARD { ALL | PLANS | TEMPORARY | TEMP }
```

Description

`DISCARD` releases internal resources associated with a database session. This command is useful for partially or fully resetting the session's state. There are several subcommands to release different types of resources. `DISCARD ALL` is not supported by Greenplum Database.

Parameters

PLANS

Releases all cached query plans, forcing re-planning to occur the next time the associated prepared statement is used.

SEQUENCES

Discards all cached sequence-related state, including any preallocated sequence values that have not yet been returned by `nextval()`. (See `CREATE SEQUENCE` for a description of preallocated sequence values.)

TEMPORARY/TEMP

Drops all temporary tables created in the current session.

ALL

Releases all temporary resources associated with the current session and resets the session to its initial state.

> **Note:** Greenplum Database does not support `DISCARD ALL` and returns a notice message if you attempt to run the command.

As an alternative, you can the run following commands to release temporary session resources:

```
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
CLOSE ALL;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD SEQUENCES;
DISCARD TEMP;
```

Compatibility

`DISCARD` is a Greenplum Database extension.

DO

Executes an anonymous code block as a transient anonymous function.

Synopsis

```
DO [ LANGUAGE lang_name ] code
```

Description

`DO` executes an anonymous code block, or in other words a transient anonymous function in a procedural language.

The code block is treated as though it were the body of a function with no parameters, returning void. It is parsed and executed a single time.

The optional `LANGUAGE` clause can appear either before or after the code block.

Anonymous blocks are procedural language structures that provide the capability to create and execute procedural code on the fly without persistently storing the code as database objects in the system catalogs. The concept of anonymous blocks is similar to UNIX shell scripts, which enable several manually entered commands to be grouped and executed as one step. As the name implies, anonymous blocks do not have a name, and for this reason they cannot be referenced from other objects. Although built dynamically, anonymous blocks can be easily stored as scripts in the operating system files for repetitive execution.

Anonymous blocks are standard procedural language blocks. They carry the syntax and obey the rules that apply to the procedural language, including declaration and scope of variables, execution, exception handling, and language usage.

The compilation and execution of anonymous blocks are combined in one step, while a user-defined function needs to be re-defined before use each time its definition changes.

Parameters

code

The procedural language code to be executed. This must be specified as a string literal, just as with the `CREATE FUNCTION` command. Use of a dollar-quoted literal is recommended. Optional keywords have no effect. These procedural languages are supported: PL/pgSQL (`plpgsql`), PL/Python (`plpythonu`), and PL/Perl (`plperl` and `plperlu`).

lang_name

The name of the procedural language that the code is written in. The default is `plpgsql`. The language must be installed on the Greenplum Database system and registered in the database.

Notes

The PL/pgSQL language is installed on the Greenplum Database system and is registered in a user created database. The PL/Python and PL/Perl languages are installed by default, but not registered. Other languages are not installed or registered. The system catalog `pg_language` contains information about the registered languages in a database.

The user must have `USAGE` privilege for the procedural language, or must be a superuser if the language is untrusted. This is the same privilege requirement as for creating a function in the language.

Anonymous blocks do not support function volatility or `EXECUTE ON` attributes.

Examples

This PL/pgSQL example grants all privileges on all views in schema *public* to role `webuser`:

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
             WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' ||
 quote_ident(r.table_name) || ' TO webuser';
    END LOOP;
END$$;
```

This PL/pgSQL example determines if a Greenplum Database user is a superuser. In the example, the anonymous block retrieves the input value from a temporary table.

```
CREATE TEMP TABLE list AS VALUES ('gpadmin') DISTRIBUTED RANDOMLY;

DO $$
DECLARE
  name TEXT := 'gpadmin' ;
  superuser TEXT := '' ;
  t1_row   pg_authid%ROWTYPE;
BEGIN
  SELECT * INTO t1_row FROM pg_authid, list
     WHERE pg_authid.rolname = name ;
  IF t1_row.rolsuper = 'f' THEN
    superuser := 'not ';
  END IF ;
  RAISE NOTICE 'user % is %a superuser', t1_row.rolname, superuser ;
END $$ LANGUAGE plpgsql ;
```

> **Note:** The example PL/pgSQL uses `SELECT` with the `INTO` clause. It is different from the SQL command `SELECT INTO`.

Compatibility

There is no `DO` statement in the SQL standard.

See Also

`CREATE LANGUAGE`

DROP AGGREGATE

Removes an aggregate function.

Synopsis

```
DROP AGGREGATE [IF EXISTS] name ( aggregate_signature ) [CASCADE | RESTRICT]
```

where *aggregate_signature* is:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname
 ] argtype [ , ... ]
```

Description

`DROP AGGREGATE` will delete an existing aggregate function. To execute this command the current user must be the owner of the aggregate function.

Parameters

`IF EXISTS`

Do not throw an error if the aggregate does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing aggregate function.

`argmode`

The mode of an argument: `IN` or `VARIADIC`. If omitted, the default is `IN`.

`argname`

The name of an argument. Note that `DROP AGGREGATE` does not actually pay any attention to argument names, since only the argument data types are needed to determine the aggregate function's identity.

`argtype`

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write `*` in place of the list of input data types. To reference an ordered-set aggregate function, write `ORDER BY` between the direct and aggregated argument specifications.

`CASCADE`

Automatically drop objects that depend on the aggregate function.

`RESTRICT`

Refuse to drop the aggregate function if any objects depend on it. This is the default.

Notes

Alternative syntaxes for referencing ordered-set aggregates are described under `ALTER AGGREGATE`.

Examples

To remove the aggregate function `myavg` for type `integer`:

```
DROP AGGREGATE myavg(integer);
```

To remove the hypothetical-set aggregate function `myrank`, which takes an arbitrary list of ordering columns and a matching list of direct arguments:

```
DROP AGGREGATE myrank(VARIADIC "any" ORDER BY VARIADIC "any");
```

Compatibility

There is no `DROP AGGREGATE` statement in the SQL standard.

See Also

`ALTER AGGREGATE`, *`CREATE AGGREGATE`*

DROP CAST

Removes a cast.

Synopsis

```
DROP CAST [IF EXISTS] (sourcetype AS targettype) [CASCADE | RESTRICT]
```

Description

`DROP CAST` will delete a previously defined cast. To be able to drop a cast, you must own the source or the target data type. These are the same privileges that are required to create a cast.

Parameters

`IF EXISTS`

Do not throw an error if the cast does not exist. A notice is issued in this case.

`sourcetype`

The name of the source data type of the cast.

`targettype`

The name of the target data type of the cast.

`CASCADE`
`RESTRICT`

These keywords have no effect since there are no dependencies on casts.

Examples

To drop the cast from type `text` to type `int`:

```
DROP CAST (text AS int);
```

Compatibility

There `DROP CAST` command conforms to the SQL standard.

See Also

`CREATE CAST`

DROP COLLATION

Removes a previously defined collation.

Synopsis

```
DROP COLLATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Parameters

`IF EXISTS`

Do not throw an error if the collation does not exist. A notice is issued in this case.

`name`

The name of the collation. The collation name can be schema-qualified.

`CASCADE`

Automatically drop objects that depend on the collation.

`RESTRICT`

Refuse to drop the collation if any objects depend on it. This is the default.

Notes

`DROP COLLATION` removes a previously defined collation. To be able to drop a collation, you must own the collation.

Examples

To drop the collation named `german`:

```
DROP COLLATION german;
```

Compatibility

The `DROP COLLATION` command conforms to the SQL standard, apart from the `IF EXISTS` option, which is a Greenplum Database extension.

See Also

`ALTER COLLATION`, *`CREATE COLLATION`*

DROP CONVERSION

Removes a conversion.

Synopsis

```
DROP CONVERSION [IF EXISTS] name [CASCADE | RESTRICT]
```

Description

`DROP CONVERSION` removes a previously defined conversion. To be able to drop a conversion, you must own the conversion.

Parameters

`IF EXISTS`

Do not throw an error if the conversion does not exist. A notice is issued in this case.

`name`

The name of the conversion. The conversion name may be schema-qualified.

`CASCADE`
`RESTRICT`

These keywords have no effect since there are no dependencies on conversions.

Examples

Drop the conversion named `myname`:

```
DROP CONVERSION myname;
```

Compatibility

There is no `DROP CONVERSION` statement in the SQL standard. The standard has `CREATE TRANSLATION` and `DROP TRANSLATION` statements that are similar to the Greenplum Database `CREATE CONVERSION` and `DROP CONVERSION` statements.

See Also

ALTER CONVERSION, *CREATE CONVERSION*

DROP DATABASE

Removes a database.

Synopsis

```
DROP DATABASE [IF EXISTS] name
```

Description

DROP DATABASE drops a database. It removes the catalog entries for the database and deletes the directory containing the data. It can only be executed by the database owner. Also, it cannot be executed while you or anyone else are connected to the target database. (Connect to `postgres` or any other database to issue this command.)

Warning: DROP DATABASE cannot be undone. Use it with care!

Parameters

IF EXISTS

Do not throw an error if the database does not exist. A notice is issued in this case.

name

The name of the database to remove.

Notes

DROP DATABASE cannot be executed inside a transaction block.

This command cannot be executed while connected to the target database. Thus, it might be more convenient to use the program `dropdb` instead, which is a wrapper around this command.

Examples

Drop the database named `testdb`:

```
DROP DATABASE testdb;
```

Compatibility

There is no DROP DATABASE statement in the SQL standard.

See Also

ALTER DATABASE, *CREATE DATABASE*

DROP DOMAIN

Removes a domain.

Synopsis

```
DROP DOMAIN [IF EXISTS] name [, ...]  [CASCADE | RESTRICT]
```

Description

`DROP DOMAIN` removes a previously defined domain. You must be the owner of a domain to drop it.

Parameters

`IF EXISTS`

Do not throw an error if the domain does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing domain.

`CASCADE`

Automatically drop objects that depend on the domain (such as table columns).

`RESTRICT`

Refuse to drop the domain if any objects depend on it. This is the default.

Examples

Drop the domain named `zipcode`:

```
DROP DOMAIN zipcode;
```

Compatibility

This command conforms to the SQL standard, except for the `IF EXISTS` option, which is a Greenplum Database extension.

See Also

`ALTER DOMAIN`, *`CREATE DOMAIN`*

DROP EXTENSION

Removes an extension from a Greenplum database.

Synopsis

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP EXTENSION` removes extensions from the database. Dropping an extension causes its component objects to be dropped as well.

> **Note:** The required supporting extension files what were installed to create the extension are not deleted. The files must be manually removed from the Greenplum Database hosts.

You must own the extension to use `DROP EXTENSION`.

This command fails if any of the extension objects are in use in the database. For example, if a table is defined with columns of the extension type. Add the `CASCADE` option to forcibly remove those dependent objects.

> **Important:** Before issuing a `DROP EXTENSION` with the `CASCADE` keyword, you should be aware of all object that depend on the extension to avoid unintended consequences.

Parameters

`IF EXISTS`

Do not throw an error if the extension does not exist. A notice is issued.

name

The name of an installed extension.

CASCADE

Automatically drop objects that depend on the extension, and in turn all objects that depend on those objects. See the PostgreSQL information about *Dependency Tracking*.

RESTRICT

Refuse to drop an extension if any objects depend on it, other than the extension member objects. This is the default.

Compatibility

`DROP EXTENSION` is a Greenplum Database extension.

See Also

`CREATE EXTENSION`, *`ALTER EXTENSION`*

DROP EXTERNAL TABLE

Removes an external table definition.

Synopsis

```
DROP EXTERNAL [WEB] TABLE [IF EXISTS] name [CASCADE | RESTRICT]
```

Description

`DROP EXTERNAL TABLE` drops an existing external table definition from the database system. The external data sources or files are not deleted. To execute this command you must be the owner of the external table.

Parameters

WEB

Optional keyword for dropping external web tables.

IF EXISTS

Do not throw an error if the external table does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing external table.

CASCADE

Automatically drop objects that depend on the external table (such as views).

RESTRICT

Refuse to drop the external table if any objects depend on it. This is the default.

Examples

Remove the external table named `staging` if it exists:

```
DROP EXTERNAL TABLE IF EXISTS staging;
```

Compatibility

There is no `DROP EXTERNAL TABLE` statement in the SQL standard.

See Also

`CREATE EXTERNAL TABLE`

DROP FOREIGN DATA WRAPPER

Removes a foreign-data wrapper.

Synopsis

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

`DROP FOREIGN DATA WRAPPER` removes an existing foreign-data wrapper from the current database. A foreign-data wrapper may be removed only by its owner.

Parameters

`IF EXISTS`

Do not throw an error if the foreign-data wrapper does not exist. Greenplum Database issues a notice in this case.

`name`

The name of an existing foreign-data wrapper.

`CASCADE`

Automatically drop objects that depend on the foreign-data wrapper (such as servers).

`RESTRICT`

Refuse to drop the foreign-data wrapper if any object depends on it. This is the default.

Examples

Drop the foreign-data wrapper named `dbi`:

```
DROP FOREIGN DATA WRAPPER dbi;
```

Compatibility

`DROP FOREIGN DATA WRAPPER` conforms to ISO/IEC 9075-9 (SQL/MED). The `IF EXISTS` clause is a Greenplum Database extension.

See Also

`CREATE FOREIGN DATA WRAPPER`, *`ALTER FOREIGN DATA WRAPPER`*

DROP FOREIGN TABLE

Removes a foreign table.

Synopsis

```
DROP FOREIGN TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP FOREIGN TABLE` removes an existing foreign table. Only the owner of a foreign table can remove it.

Parameters

`IF EXISTS`

Do not throw an error if the foreign table does not exist. Greenplum Database issues a notice in this case.

`name`

The name (optionally schema-qualified) of the foreign table to drop.

`CASCADE`

Automatically drop objects that depend on the foreign table (such as views).

`RESTRICT`

Refuse to drop the foreign table if any objects depend on it. This is the default.

Examples

Drop the foreign tables named `films` and `distributors`:

```
DROP FOREIGN TABLE films, distributors;
```

Compatibility

`DROP FOREIGN TABLE` conforms to ISO/IEC 9075-9 (SQL/MED), except that the standard only allows one foreign table to be dropped per command. The `IF EXISTS` clause is a Greenplum Database extension.

See Also

`ALTER FOREIGN TABLE`, *`CREATE FOREIGN TABLE`*

DROP FUNCTION

Removes a function.

Synopsis

```
DROP FUNCTION [IF EXISTS] name ( [ [argmode] [argname] argtype
    [, ...] ] ) [CASCADE | RESTRICT]
```

Description

`DROP FUNCTION` removes the definition of an existing function. To execute this command the user must be the owner of the function. The argument types to the function must be specified, since several different functions may exist with the same name and different argument lists.

Parameters

`IF EXISTS`

Do not throw an error if the function does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: either `IN`, `OUT`, `INOUT`, or `VARIADIC`. If omitted, the default is IN. Note that `DROP FUNCTION` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the `IN`, `INOUT`, and `VARIADIC` arguments.

argname

The name of an argument. Note that `DROP FUNCTION` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

`CASCADE`

Automatically drop objects that depend on the function such as operators.

`RESTRICT`

Refuse to drop the function if any objects depend on it. This is the default.

Examples

Drop the square root function:

```
DROP FUNCTION sqrt(integer);
```

Compatibility

A `DROP FUNCTION` statement is defined in the SQL standard, but it is not compatible with this command.

See Also

`CREATE FUNCTION`, *`ALTER FUNCTION`*

DROP GROUP

Removes a database role.

Synopsis

```
DROP GROUP [IF EXISTS] name [, ...]
```

Description

`DROP GROUP` is an alias for `DROP ROLE`. See *`DROP ROLE`* for more information.

Compatibility

There is no `DROP GROUP` statement in the SQL standard.

See Also

`DROP ROLE`

DROP INDEX

Removes an index.

Synopsis

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] name [, ...] [ CASCADE |
 RESTRICT ]
```

Description

`DROP INDEX` drops an existing index from the database system. To execute this command you must be the owner of the index.

Parameters

`CONCURRENTLY`

Drop the index without locking out concurrent selects, inserts, updates, and deletes on the index's table. A normal `DROP INDEX` acquires an exclusive lock on the table, blocking other accesses until the index drop can be completed. With this option, the command instead waits until conflicting transactions have completed.

There are several caveats to be aware of when using this option. Only one index name can be specified, and the `CASCADE` option is not supported. (Thus, an index that supports a `UNIQUE` or `PRIMARY KEY` constraint cannot be dropped this way.) Also, regular `DROP INDEX` commands can be performed within a transaction block, but `DROP INDEX CONCURRENTLY` cannot.

`IF EXISTS`

Do not throw an error if the index does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing index.

`CASCADE`

Automatically drop objects that depend on the index.

`RESTRICT`

Refuse to drop the index if any objects depend on it. This is the default.

Examples

Remove the index `title_idx`:

```
DROP INDEX title_idx;
```

Compatibility

`DROP INDEX` is a Greenplum Database language extension. There are no provisions for indexes in the SQL standard.

See Also

`ALTER INDEX`, *`CREATE INDEX`*, *`REINDEX`*

DROP LANGUAGE

Removes a procedural language.

Synopsis

```
DROP [PROCEDURAL] LANGUAGE [IF EXISTS] name [CASCADE | RESTRICT]
```

Description

`DROP LANGUAGE` will remove the definition of the previously registered procedural language. You must be a superuser or owner of the language to drop a language.

Parameters

PROCEDURAL

Optional keyword - has no effect.

IF EXISTS

Do not throw an error if the language does not exist. A notice is issued in this case.

name

The name of an existing procedural language. For backward compatibility, the name may be enclosed by single quotes.

CASCADE

Automatically drop objects that depend on the language (such as functions written in that language).

RESTRICT

Refuse to drop the language if any objects depend on it. This is the default.

Examples

Remove the procedural language `plsample`:

```
DROP LANGUAGE plsample;
```

Compatibility

There is no `DROP LANGUAGE` statement in the SQL standard.

See Also

ALTER LANGUAGE, *CREATE LANGUAGE*

DROP MATERIALIZED VIEW

Removes a materialized view.

Synopsis

```
DROP MATERIALIZED VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP MATERIALIZED VIEW` drops an existing materialized view. To execute this command, you must be the owner of the materialized view.

Parameters

`IF EXISTS`

Do not throw an error if the materialized view does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of a materialized view to be dropped.

`CASCADE`

Automatically drop objects that depend on the materialized view (such as other materialized views, or regular views).

`RESTRICT`

Refuse to drop the materialized view if any objects depend on it. This is the default.

Examples

This command removes the materialized view called `order_summary`.

```
DROP MATERIALIZED VIEW order_summary;
```

Compatibility

`DROP MATERIALIZED VIEW` is a Greenplum Database extension of the SQL standard.

See Also

`ALTER MATERIALIZED VIEW`, *`CREATE MATERIALIZED VIEW`*, *`REFRESH MATERIALIZED VIEW`*

DROP OPERATOR

Removes an operator.

Synopsis

```
DROP OPERATOR [IF EXISTS] name ( {lefttype | NONE} ,
    {righttype | NONE} ) [CASCADE | RESTRICT]
```

Description

`DROP OPERATOR` drops an existing operator from the database system. To execute this command you must be the owner of the operator.

Parameters

`IF EXISTS`

Do not throw an error if the operator does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing operator.

`lefttype`

The data type of the operator's left operand; write `NONE` if the operator has no left operand.

`righttype`

The data type of the operator's right operand; write `NONE` if the operator has no right operand.

CASCADE

Automatically drop objects that depend on the operator.

RESTRICT

Refuse to drop the operator if any objects depend on it. This is the default.

Examples

Remove the power operator `a^b` for type `integer`:

```
DROP OPERATOR ^ (integer, integer);
```

Remove the left unary bitwise complement operator `~b` for type `bit`:

```
DROP OPERATOR ~ (none, bit);
```

Remove the right unary factorial operator `x!` for type `bigint`:

```
DROP OPERATOR ! (bigint, none);
```

Compatibility

There is no `DROP OPERATOR` statement in the SQL standard.

See Also

`ALTER OPERATOR`, *`CREATE OPERATOR`*

DROP OPERATOR CLASS

Removes an operator class.

Synopsis

```
DROP OPERATOR CLASS [IF EXISTS] name USING index_method [CASCADE | RESTRICT]
```

Description

`DROP OPERATOR` drops an existing operator class. To execute this command you must be the owner of the operator class.

Parameters

IF EXISTS

Do not throw an error if the operator class does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing operator class.

index_method

The name of the index access method the operator class is for.

CASCADE

Automatically drop objects that depend on the operator class.

RESTRICT

Refuse to drop the operator class if any objects depend on it. This is the default.

Examples

Remove the B-tree operator class `widget_ops`:

```
DROP OPERATOR CLASS widget_ops USING btree;
```

This command will not succeed if there are any existing indexes that use the operator class. Add `CASCADE` to drop such indexes along with the operator class.

Compatibility

There is no `DROP OPERATOR CLASS` statement in the SQL standard.

See Also

`ALTER OPERATOR CLASS`, *`CREATE OPERATOR CLASS`*

DROP OPERATOR FAMILY

Removes an operator family.

Synopsis

```
DROP OPERATOR FAMILY [IF EXISTS] name USING index_method [CASCADE |
 RESTRICT]
```

Description

`DROP OPERATOR FAMILY` drops an existing operator family. To execute this command you must be the owner of the operator family.

`DROP OPERATOR FAMILY` includes dropping any operator classes contained in the family, but it does not drop any of the operators or functions referenced by the family. If there are any indexes depending on operator classes within the family, you will need to specify `CASCADE` for the drop to complete.

Parameters

`IF EXISTS`

Do not throw an error if the operator family does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing operator family.

`index_method`

The name of the index access method the operator family is for.

`CASCADE`

Automatically drop objects that depend on the operator family.

`RESTRICT`

Refuse to drop the operator family if any objects depend on it. This is the default.

Examples

Remove the B-tree operator family `float_ops`:

```
DROP OPERATOR FAMILY float_ops USING btree;
```

This command will not succeed if there are any existing indexes that use the operator family. Add CASCADE to drop such indexes along with the operator family.

Compatibility

There is no DROP OPERATOR FAMILY statement in the SQL standard.

See Also

ALTER OPERATOR FAMILY, *CREATE OPERATOR FAMILY*, *ALTER OPERATOR CLASS*, *CREATE OPERATOR CLASS*, *DROP OPERATOR CLASS*

DROP OWNED

Removes database objects owned by a database role.

Synopsis

```
DROP OWNED BY name [, ...] [CASCADE | RESTRICT]
```

Description

DROP OWNED drops all the objects in the current database that are owned by one of the specified roles. Any privileges granted to the given roles on objects in the current database or on shared objects (databases, tablespaces) will also be revoked.

Parameters

name

The name of a role whose objects will be dropped, and whose privileges will be revoked.

CASCADE

Automatically drop objects that depend on the affected objects.

RESTRICT

Refuse to drop the objects owned by a role if any other database objects depend on one of the affected objects. This is the default.

Notes

DROP OWNED is often used to prepare for the removal of one or more roles. Because DROP OWNED only affects the objects in the current database, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed.

Using the CASCADE option may make the command recurse to objects owned by other users.

The *REASSIGN OWNED* command is an alternative that reassigns the ownership of all the database objects owned by one or more roles. However, REASSIGN OWNED does not deal with privileges for other objects.

Examples

Remove any database objects owned by the role named sally:

```
DROP OWNED BY sally;
```

Compatibility

The DROP OWNED command is a Greenplum Database extension.

See Also

REASSIGN OWNED, *DROP ROLE*

DROP PROTOCOL

Removes a external table data access protocol from a database.

Synopsis

```
DROP PROTOCOL [IF EXISTS] name
```

Description

DROP PROTOCOL removes the specified protocol from a database. A protocol name can be specified in the CREATE EXTERNAL TABLE command to read data from or write data to an external data source.

You must be a superuser or the protocol owner to drop a protocol.

> **Warning:** If you drop a data access prococol, external tables that have been defined with the protocol will no longer be able to access the external data source.

Parameters

IF EXISTS

Do not throw an error if the protocol does not exist. A notice is issued in this case.

name

The name of an existing data access protocol.

Notes

If you drop a data access protocol, the call handlers that defined in the database that are associated with the protocol are not dropped. You must drop the functions manually.

Shared libraries that were used by the protocol should also be removed from the Greenplum Database hosts.

Compatibility

DROP PROTOCOL is a Greenplum Database extension.

See Also

CREATE EXTERNAL TABLE, *CREATE PROTOCOL*

DROP RESOURCE GROUP

Removes a resource group.

Synopsis

```
DROP RESOURCE GROUP group_name
```

Description

This command removes a resource group from Greenplum Database. Only a superuser can drop a resource group. When you drop a resource group, the memory and CPU resources reserved by the group are returned to Greenplum Database.

To drop a role resource group, the group cannot be assigned to any roles, nor can it have any statements pending or running in the group. If you drop a resource group that you created for an external component, the behavior is determined by the external component. For example, dropping a resource group that you assigned to a PL/Container runtime kills running containers in the group.

You cannot drop the pre-defined `admin_group` and `default_group` resource groups.

Parameters

group_name

The name of the resource group to remove.

Notes

You cannot submit a `DROP RESOURCE GROUP` command in an explicit transaction or sub-transaction.

Use *`ALTER ROLE`* to remove a resource group assigned to a specific user/role.

Perform the following query to view all of the currently active queries for all resource groups:

```
SELECT usename, query, waiting, pid,
    rsgid, rsgname, rsgqueueduration
  FROM pg_stat_activity;
```

To view the resource group assignments, perform the following query on the `pg_roles` and `pg_resgroup` system catalog tables:

```
SELECT rolname, rsgname
  FROM pg_roles, pg_resgroup
  WHERE pg_roles.rolresgroup=pg_resgroup.oid;
```

Examples

Remove the resource group assigned to a role. This operation then assigns the default resource group `default_group` to the role:

```
ALTER ROLE bob RESOURCE GROUP NONE;
```

Remove the resource group named `adhoc`:

```
DROP RESOURCE GROUP adhoc;
```

Compatibility

The `DROP RESOURCE GROUP` statement is a Greenplum Database extension.

See Also

`ALTER RESOURCE GROUP`, *`CREATE RESOURCE GROUP`*, *`ALTER ROLE`*

DROP RESOURCE QUEUE

Removes a resource queue.

Synopsis

```
DROP RESOURCE QUEUE queue_name
```

Description

This command removes a resource queue from Greenplum Database. To drop a resource queue, the queue cannot have any roles assigned to it, nor can it have any statements waiting in the queue. Only a superuser can drop a resource queue.

Parameters

queue_name

The name of a resource queue to remove.

Notes

Use *ALTER ROLE* to remove a user from a resource queue.

To see all the currently active queries for all resource queues, perform the following query of the `pg_locks` table joined with the `pg_roles` and `pg_resqueue` tables:

```
SELECT rolname, rsqname, locktype, objid, pid,
mode, granted FROM pg_roles, pg_resqueue, pg_locks WHERE
pg_roles.rolresqueue=pg_locks.objid AND
pg_locks.objid=pg_resqueue.oid;
```

To see the roles assigned to a resource queue, perform the following query of the `pg_roles` and `pg_resqueue` system catalog tables:

```
SELECT rolname, rsqname FROM pg_roles, pg_resqueue WHERE
pg_roles.rolresqueue=pg_resqueue.oid;
```

Examples

Remove a role from a resource queue (and move the role to the default resource queue, `pg_default`):

```
ALTER ROLE bob RESOURCE QUEUE NONE;
```

Remove the resource queue named `adhoc`:

```
DROP RESOURCE QUEUE adhoc;
```

Compatibility

The `DROP RESOURCE QUEUE` statement is a Greenplum Database extension.

See Also

ALTER RESOURCE QUEUE, *CREATE RESOURCE QUEUE*, *ALTER ROLE*

DROP ROLE

Removes a database role.

Synopsis

```
DROP ROLE [IF EXISTS] name [, ...]
```

Description

`DROP ROLE` removes the specified role(s). To drop a superuser role, you must be a superuser yourself. To drop non-superuser roles, you must have `CREATEROLE` privilege.

A role cannot be removed if it is still referenced in any database; an error will be raised if so. Before dropping the role, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the role has been granted on other objects. The *`REASSIGN OWNED`* and *`DROP OWNED`* commands can be useful for this purpose.

However, it is not necessary to remove role memberships involving the role; `DROP ROLE` automatically revokes any memberships of the target role in other roles, and of other roles in the target role. The other roles are not dropped nor otherwise affected.

Parameters

`IF EXISTS`

Do not throw an error if the role does not exist. A notice is issued in this case.

`name`

The name of the role to remove.

Examples

Remove the roles named `sally` and `bob`:

```
DROP ROLE sally, bob;
```

Compatibility

The SQL standard defines `DROP ROLE`, but it allows only one role to be dropped at a time, and it specifies different privilege requirements than Greenplum Database uses.

See Also

`REASSIGN OWNED`, *`DROP OWNED`*, *`CREATE ROLE`*, *`ALTER ROLE`*, *`SET ROLE`*

DROP RULE

Removes a rewrite rule.

Synopsis

```
DROP RULE [IF EXISTS] name ON table_name [CASCADE | RESTRICT]
```

Description

`DROP RULE` drops a rewrite rule from a table or view.

Parameters

`IF EXISTS`

Do not throw an error if the rule does not exist. A notice is issued in this case.

`name`

The name of the rule to remove.

`table_name`

The name (optionally schema-qualified) of the table or view that the rule applies to.

CASCADE

Automatically drop objects that depend on the rule.

RESTRICT

Refuse to drop the rule if any objects depend on it. This is the default.

Examples

Remove the rewrite rule `sales_2006` on the table `sales`:

```
DROP RULE sales_2006 ON sales;
```

Compatibility

`DROP RULE` is a Greenplum Database language extension, as is the entire query rewrite system.

See Also

`ALTER RULE`, *`CREATE RULE`*

DROP SCHEMA

Removes a schema.

Synopsis

```
DROP SCHEMA [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP SCHEMA` removes schemas from the database. A schema can only be dropped by its owner or a superuser. Note that the owner can drop the schema (and thereby all contained objects) even if he does not own some of the objects within the schema.

Parameters

IF EXISTS

Do not throw an error if the schema does not exist. A notice is issued in this case.

name

The name of the schema to remove.

CASCADE

Automatically drops any objects contained in the schema (tables, functions, etc.).

RESTRICT

Refuse to drop the schema if it contains any objects. This is the default.

Examples

Remove the schema `mystuff` from the database, along with everything it contains:

```
DROP SCHEMA mystuff CASCADE;
```

Compatibility

`DROP SCHEMA` is fully conforming with the SQL standard, except that the standard only allows one schema to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

CREATE SCHEMA, *ALTER SCHEMA*

DROP SEQUENCE

Removes a sequence.

Synopsis

```
DROP SEQUENCE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP SEQUENCE` removes a sequence generator table. You must own the sequence to drop it (or be a superuser).

Parameters

`IF EXISTS`

Do not throw an error if the sequence does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the sequence to remove.

`CASCADE`

Automatically drop objects that depend on the sequence.

`RESTRICT`

Refuse to drop the sequence if any objects depend on it. This is the default.

Examples

Remove the sequence `myserial`:

```
DROP SEQUENCE myserial;
```

Compatibility

`DROP SEQUENCE` is fully conforming with the SQL standard, except that the standard only allows one sequence to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

ALTER SEQUENCE, *CREATE SEQUENCE*

DROP SERVER

Removes a foreign server descriptor.

Synopsis

```
DROP SERVER [ IF EXISTS ] servername [ CASCADE | RESTRICT ]
```

Description

DROP SERVER removes an existing foreign server descriptor. The user executing this command must be the owner of the server.

Parameters

IF EXISTS

Do not throw an error if the server does not exist. Greenplum Database issues a notice in this case.

servername

The name of an existing server.

CASCADE

Automatically drop objects that depend on the server (such as user mappings).

RESTRICT

Refuse to drop the server if any object depends on it. This is the default.

Examples

Drop the server named `foo` if it exists:

```
DROP SERVER IF EXISTS foo;
```

Compatibility

DROP SERVER conforms to ISO/IEC 9075-9 (SQL/MED). The IF EXISTS clause is a Greenplum Database extension.

See Also

CREATE SERVER, *ALTER SERVER*

DROP TABLE

Removes a table.

Synopsis

```
DROP TABLE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

DROP TABLE removes tables from the database. Only the table owner, the schema owner, and superuser can drop a table. To empty a table of rows without removing the table definition, use DELETE or TRUNCATE.

DROP TABLE always removes any indexes, rules, triggers, and constraints that exist for the target table. However, to drop a table that is referenced by a view, CASCADE must be specified. CASCADE will remove a dependent view entirely.

Parameters

IF EXISTS

Do not throw an error if the table does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the table to remove.

CASCADE

Automatically drop objects that depend on the table (such as views).

RESTRICT

Refuse to drop the table if any objects depend on it. This is the default.

Examples

Remove the table `mytable`:

```
DROP TABLE mytable;
```

Compatibility

`DROP TABLE` is fully conforming with the SQL standard, except that the standard only allows one table to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

`CREATE TABLE`, *`ALTER TABLE`*, *`TRUNCATE`*

DROP TABLESPACE

Removes a tablespace.

Synopsis

```
DROP TABLESPACE [IF EXISTS] tablespacename
```

Description

`DROP TABLESPACE` removes a tablespace from the system.

A tablespace can only be dropped by its owner or a superuser. The tablespace must be empty of all database objects before it can be dropped. It is possible that objects in other databases may still reside in the tablespace even if no objects in the current database are using the tablespace. Also, if the tablespace is listed in the *temp_tablespaces* setting of any active session, `DROP TABLESPACE` might fail due to temporary files residing in the tablespace.

Parameters

IF EXISTS

Do not throw an error if the tablespace does not exist. A notice is issued in this case.

tablespacename

The name of the tablespace to remove.

Notes

Run `DROP TABLESPACE` during a period of low activity to avoid issues due to concurrent creation of tables and temporary objects. When a tablespace is dropped, there is a small window in which a table could be created in the tablespace that is currently being dropped. If this occurs, Greenplum Database returns a warning. This is an example of the `DROP TABLESPACE` warning.

```
testdb=# DROP TABLESPACE mytest;
```

```
WARNING:  tablespace with oid "16415" is not empty  (seg1
 192.168.8.145:25433 pid=29023)
WARNING:  tablespace with oid "16415" is not empty  (seg0
 192.168.8.145:25432 pid=29022)
WARNING:  tablespace with oid "16415" is not empty
DROP TABLESPACE
```

The table data in the tablespace directory is not dropped. You can use the *ALTER TABLE* command to change the tablespace defined for the table and move the data to an existing tablespace.

Examples

Remove the tablespace `mystuff`:

```
DROP TABLESPACE mystuff;
```

Compatibility

`DROP TABLESPACE` is a Greenplum Database extension.

See Also

CREATE TABLESPACE, *ALTER TABLESPACE*

DROP TEXT SEARCH CONFIGURATION

Removes a text search configuration.

Synopsis

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

DROP TEXT SEARCH CONFIGURATION drops an existing text search configuration. To execute this command you must be the owner of the configuration.

Parameters

`IF EXISTS`

Do not throw an error if the text search configuration does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing text search configuration.

`CASCADE`

Automatically drop objects that depend on the text search configuration.

`RESTRICT`

Refuse to drop the text search configuration if any objects depend on it. This is the default.

Examples

Remove the text search configuration my_english:

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

This command will not succeed if there are any existing indexes that reference the configuration in `to_tsvector` calls. Add `CASCADE` to drop such indexes along with the text search configuration.

Compatibility

There is no `DROP TEXT SEARCH CONFIGURATION` statement in the SQL standard.

See Also

`ALTER TEXT SEARCH CONFIGURATION`, *`CREATE TEXT SEARCH CONFIGURATION`*

DROP TEXT SEARCH DICTIONARY

Removes a text search dictionary.

Synopsis

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

`DROP TEXT SEARCH DICTIONARY` drops an existing text search dictionary. To execute this command you must be the owner of the dictionary.

Parameters

`IF EXISTS`

Do not throw an error if the text search dictionary does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing text search dictionary.

`CASCADE`

Automatically drop objects that depend on the text search dictionary.

`RESTRICT`

Refuse to drop the text search dictionary if any objects depend on it. This is the default.

Examples

Remove the text search dictionary `english`:

```
DROP TEXT SEARCH DICTIONARY english;
```

This command will not succeed if there are any existing text search configurations that use the dictionary. Add `CASCADE` to drop such configurations along with the dictionary.

Compatibility

There is no `CREATE TEXT SEARCH DICTIONARY` statement in the SQL standard.

See Also

`ALTER TEXT SEARCH DICTIONARY`, *`CREATE TEXT SEARCH DICTIONARY`*

DROP TEXT SEARCH PARSER

Description

Remove a text search parser.

Synopsis

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

`DROP TEXT SEARCH PARSER` drops an existing text search parser. You must be a superuser to use this command.

Parameters

`IF EXISTS`

Do not throw an error if the text search parser does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing text search parser.

`CASCADE`

Automatically drop objects that depend on the text search parser.

`RESTRICT`

Refuse to drop the text search parser if any objects depend on it. This is the default.

Examples

Remove the text search parser `my_parser`:

```
DROP TEXT SEARCH PARSER my_parser;
```

This command will not succeed if there are any existing text search configurations that use the parser. Add `CASCADE` to drop such configurations along with the parser.

Compatibility

There is no `DROP TEXT SEARCH PARSER` statement in the SQL standard.

See Also

`ALTER TEXT SEARCH PARSER`, *`CREATE TEXT SEARCH PARSER`*

DROP TEXT SEARCH TEMPLATE

Description

Removes a text search template.

Synopsis

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

`DROP TEXT SEARCH TEMPLATE` drops an existing text search template. You must be a superuser to use this command.

You must be a superuser to use `ALTER TEXT SEARCH TEMPLATE`.

Parameters

`IF EXISTS`

Do not throw an error if the text search template does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing text search template.

`CASCADE`

Automatically drop objects that depend on the text search template.

`RESTRICT`

Refuse to drop the text search template if any objects depend on it. This is the default.

Compatibility

There is no `DROP TEXT SEARCH TEMPLATE` statement in the SQL standard.

See Also

`ALTER TEXT SEARCH TEMPLATE`, *`CREATE TEXT SEARCH TEMPLATE`*

DROP TYPE

Removes a data type.

Synopsis

```
DROP TYPE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP TYPE` will remove a user-defined data type. Only the owner of a type can remove it.

Parameters

`IF EXISTS`

Do not throw an error if the type does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of the data type to remove.

`CASCADE`

Automatically drop objects that depend on the type (such as table columns, functions, operators).

`RESTRICT`

Refuse to drop the type if any objects depend on it. This is the default.

Examples

Remove the data type box;

```
DROP TYPE box;
```

Compatibility

This command is similar to the corresponding command in the SQL standard, apart from the `IF EXISTS` option, which is a Greenplum Database extension. But note that much of the `CREATE TYPE` command and the data type extension mechanisms in Greenplum Database differ from the SQL standard.

See Also

ALTER TYPE, *CREATE TYPE*

DROP USER

Removes a database role.

Synopsis

```
DROP USER [IF EXISTS] name [, ...]
```

Description

`DROP USER` is an alias for *DROP ROLE*. See *DROP ROLE* for more information.

Compatibility

There is no `DROP USER` statement in the SQL standard. The SQL standard leaves the definition of users to the implementation.

See Also

DROP ROLE

DROP USER MAPPING

Removes a user mapping for a foreign server.

Synopsis

```
DROP USER MAPPING [ IF EXISTS ] { username | USER | CURRENT_USER | PUBLIC }
    SERVER servername
```

Description

`DROP USER MAPPING` removes an existing user mapping from a foreign server. To execute this command, the current user must be the owner of the server containing the mapping.

Parameters

`IF EXISTS`

Do not throw an error if the user mapping does not exist. Greenplum Database issues a notice in this case.

username

User name of the mapping. `CURRENT_USER` and `USER` match the name of the current user. `PUBLIC` is used to match all present and future user names in the system.

servername

Server name of the user mapping.

Examples

Drop the user mapping named `bob`, server `foo` if it exists:

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

Compatibility

`DROP SERVER` conforms to ISO/IEC 9075-9 (SQL/MED). The `IF EXISTS` clause is a Greenplum Database extension.

See Also

`CREATE USER MAPPING`, *`ALTER USER MAPPING`*

DROP VIEW

Removes a view.

Synopsis

```
DROP VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP VIEW` will remove an existing view. Only the owner of a view can remove it.

Parameters

`IF EXISTS`

Do not throw an error if the view does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the view to remove.

`CASCADE`

Automatically drop objects that depend on the view (such as other views).

`RESTRICT`

Refuse to drop the view if any objects depend on it. This is the default.

Examples

Remove the view `topten`;

```
DROP VIEW topten;
```

Compatibility

`DROP VIEW` is fully conforming with the SQL standard, except that the standard only allows one view to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

`CREATE VIEW`

END

Commits the current transaction.

Synopsis

```
END [WORK | TRANSACTION]
```

Description

`END` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs. This command is a Greenplum Database extension that is equivalent to *`COMMIT`*.

Parameters

`WORK`
`TRANSACTION`

Optional keywords. They have no effect.

Examples

Commit the current transaction:

```
END;
```

Compatibility

`END` is a Greenplum Database extension that provides functionality equivalent to *`COMMIT`*, which is specified in the SQL standard.

See Also

`BEGIN`, *`ROLLBACK`*, *`COMMIT`*

EXECUTE

Executes a prepared SQL statement.

Synopsis

```
EXECUTE name [ (parameter [, ...] ) ]
```

Description

`EXECUTE` is used to execute a previously prepared statement. Since prepared statements only exist for the duration of a session, the prepared statement must have been created by a `PREPARE` statement executed earlier in the current session.

If the PREPARE statement that created the statement specified some parameters, a compatible set of parameters must be passed to the EXECUTE statement, or else an error is raised. Note that (unlike functions) prepared statements are not overloaded based on the type or number of their parameters; the name of a prepared statement must be unique within a database session.

For more information on the creation and usage of prepared statements, see PREPARE.

Parameters

name

The name of the prepared statement to execute.

parameter

The actual value of a parameter to the prepared statement. This must be an expression yielding a value that is compatible with the data type of this parameter, as was determined when the prepared statement was created.

Examples

Create a prepared statement for an INSERT statement, and then execute it:

```
PREPARE fooplan (int, text, bool, numeric) AS INSERT INTO
foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Compatibility

The SQL standard includes an EXECUTE statement, but it is only for use in embedded SQL. This version of the EXECUTE statement also uses a somewhat different syntax.

See Also

DEALLOCATE, *PREPARE*

EXPLAIN

Shows the query plan of a statement.

Synopsis

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ANALYZE] [VERBOSE] statement
```

where *option* can be one of:

```
    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    BUFFERS [ boolean ]
    TIMING [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

Description

EXPLAIN displays the query plan that the Greenplum or Postgres Planner generates for the supplied statement. Query plans are a tree plan of nodes. Each node in the plan represents a single operation, such as table scan, join, aggregation or a sort.

Plans should be read from the bottom up as each node feeds rows into the node directly above it. The bottom nodes of a plan are usually table scan operations (sequential, index or bitmap index scans). If the query requires joins, aggregations, or sorts (or other operations on the raw rows) then there will be additional nodes above the scan nodes to perform these operations. The topmost plan nodes are usually the Greenplum Database motion nodes (redistribute, explicit redistribute, broadcast, or gather motions). These are the operations responsible for moving rows between the segment instances during query processing.

The output of `EXPLAIN` has one line for each node in the plan tree, showing the basic node type plus the following cost estimates that the planner made for the execution of that plan node:

- **cost** — the planner's guess at how long it will take to run the statement (measured in cost units that are arbitrary, but conventionally mean disk page fetches). Two cost numbers are shown: the start-up cost before the first row can be returned, and the total cost to return all the rows. Note that the total cost assumes that all rows will be retrieved, which may not always be the case (if using `LIMIT` for example).
- **rows** — the total number of rows output by this plan node. This is usually less than the actual number of rows processed or scanned by the plan node, reflecting the estimated selectivity of any `WHERE` clause conditions. Ideally the top-level nodes estimate will approximate the number of rows actually returned, updated, or deleted by the query.
- **width** — total bytes of all the rows output by this plan node.

It is important to note that the cost of an upper-level node includes the cost of all its child nodes. The topmost node of the plan has the estimated total execution cost for the plan. This is this number that the planner seeks to minimize. It is also important to realize that the cost only reflects things that the query optimizer cares about. In particular, the cost does not consider the time spent transmitting result rows to the client.

`EXPLAIN ANALYZE` causes the statement to be actually executed, not only planned. The `EXPLAIN ANALYZE` plan shows the actual results along with the planner's estimates. This is useful for seeing whether the planner's estimates are close to reality. In addition to the information shown in the `EXPLAIN` plan, `EXPLAIN ANALYZE` will show the following additional information:

- The total elapsed time (in milliseconds) that it took to run the query.
- The number of *workers* (segments) involved in a plan node operation. Only segments that return rows are counted.
- The maximum number of rows returned by the segment that produced the most rows for an operation. If multiple segments produce an equal number of rows, the one with the longest *time to end* is the one chosen.
- The segment id number of the segment that produced the most rows for an operation.
- For relevant operations, the *work_mem* used by the operation. If `work_mem` was not sufficient to perform the operation in memory, the plan will show how much data was spilled to disk and how many passes over the data were required for the lowest performing segment. For example:

```
Work_mem used: 64K bytes avg, 64K bytes max (seg0).
Work_mem wanted: 90K bytes avg, 90K bytes max (seg0) to abate workfile
I/O affecting 2 workers.
[seg0] pass 0: 488 groups made from 488 rows; 263 rows written to
workfile
[seg0] pass 1: 263 groups made from 263 rows
```

- The time (in milliseconds) it took to retrieve the first row from the segment that produced the most rows, and the total time taken to retrieve all rows from that segment. The *<time> to first row* may be omitted if it is the same as the *<time> to end*.

  Important: Keep in mind that the statement is actually executed when `ANALYZE` is used. Although `EXPLAIN ANALYZE` will discard any output that a `SELECT` would return, other side effects of the

statement will happen as usual. If you wish to use `EXPLAIN ANALYZE` on a DML statement without letting the command affect your data, use this approach:

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

Only the `ANALYZE` and `VERBOSE` options can be specified, and only in that order, without surrounding the option list in parentheses.

Parameters

`ANALYZE`

Carry out the command and show the actual run times and other statistics. This parameter defaults to `FALSE` if you omit it; specify `ANALYZE true` to enable it.

`VERBOSE`

Display additional information regarding the plan. Specifically, include the output column list for each node in the plan tree, schema-qualify table and function names, always label variables in expressions with their range table alias, and always print the name of each trigger for which statistics are displayed. This parameter defaults to `FALSE`if you omit it; specify `VERBOSE true` to enable it.

`COSTS`

Include information on the estimated startup and total cost of each plan node, as well as the estimated number of rows and the estimated width of each row. This parameter defaults to `TRUE` if you omit it; specify `COSTS false` to disable it.

`BUFFERS`

Include information on buffer usage. Specifically, include the number of shared blocks hit, read, dirtied, and written, the number of local blocks hit, read, dirtied, and written, and the number of temp blocks read and written. A *hit* means that a read was avoided because the block was found already in cache when needed. Shared blocks contain data from regular tables and indexes; local blocks contain data from temporary tables and indexes; while temp blocks contain short-term working data used in sorts, hashes, Materialize plan nodes, and similar cases. The number of blocks *dirtied* indicates the number of previously unmodified blocks that were changed by this query; while the number of blocks *written* indicates the number of previously-dirtied blocks evicted from cache by this backend during query processing. The number of blocks shown for an upper-level node includes those used by all its child nodes. In text format, only non-zero values are printed. This parameter may only be used when `ANALYZE` is also enabled. This parameter defaults to `FALSE` if you omit it; specify `BUFFERS true` to enable it.

`TIMING`

Include actual startup time and time spent in each node in the output. The overhead of repeatedly reading the system clock can slow down the query significantly on some systems, so it may be useful to set this parameter to `FALSE` when only actual row counts, and not exact times, are needed. Run time of the entire statement is always measured, even when node-level timing is turned off with this option. This parameter may only be used when `ANALYZE` is also enabled. It defaults to `TRUE`.

`FORMAT`

Specify the output format, which can be `TEXT`, `XML`, `JSON`, or `YAML`. Non-text output contains the same information as the text output format, but is easier for programs to parse. This parameter defaults to `TEXT`.

`boolean`

Specifies whether the selected option should be turned on or off. You can write `TRUE`, `ON`, or `1` to enable the option, and `FALSE`, `OFF`, or `0` to disable it. The boolean value can also be omitted, in which case `TRUE` is assumed.

statement

Any `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `VALUES`, `EXECUTE`, `DECLARE`, or `CREATE TABLE AS` statement, whose execution plan you wish to see.

Notes

In order to allow the query optimizer to make reasonably informed decisions when optimizing queries, the `ANALYZE` statement should be run to record statistics about the distribution of data within the table. If you have not done this (or if the statistical distribution of the data in the table has changed significantly since the last time `ANALYZE` was run), the estimated costs are unlikely to conform to the real properties of the query, and consequently an inferior query plan may be chosen.

An SQL statement that is run during the execution of an `EXPLAIN ANALYZE` command is excluded from Greenplum Database resource queues.

For more information about query profiling, see "Query Profiling" in the *Greenplum Database Administrator Guide*. For more information about resource queues, see "Resource Management with Resource Queues" in the *Greenplum Database Administrator Guide*.

Examples

To illustrate how to read an `EXPLAIN` query plan, consider the following example for a very simple query:

```
EXPLAIN SELECT * FROM names WHERE name = 'Joelle';
                                 QUERY PLAN
-------------------------------------------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)  (cost=0.00..431.27 rows=1
 width=58)
   ->  Seq Scan on names  (cost=0.00..431.27 rows=1 width=58)
         Filter: (name = 'Joelle'::text)
 Optimizer: Pivotal Optimizer (GPORCA) version 3.23.0
(4 rows)
```

If we read the plan from the bottom up, the query optimizer starts by doing a sequential scan of the `names` table. Notice that the `WHERE` clause is being applied as a *filter* condition. This means that the scan operation checks the condition for each row it scans, and outputs only the ones that pass the condition.

The results of the scan operation are passed up to a *gather motion* operation. In Greenplum Database, a gather motion is when segments send rows up to the master. In this case we have 3 segment instances sending to 1 master instance (3:1). This operation is working on `slice1` of the parallel query execution plan. In Greenplum Database a query plan is divided into *slices* so that portions of the query plan can be worked on in parallel by the segments.

The estimated startup cost for this plan is `00.00` (no cost) and a total cost of `431.27`. The planner is estimating that this query will return one row.

Here is the same query, with cost estimates suppressed:

```
EXPLAIN (COSTS FALSE) SELECT * FROM names WHERE name = 'Joelle';
                QUERY PLAN
------------------------------------------
 Gather Motion 3:1  (slice1; segments: 3)
   ->  Seq Scan on names
         Filter: (name = 'Joelle'::text)
 Optimizer: Pivotal Optimizer (GPORCA) version 3.23.0
(4 rows)
```

Here is the same query, with JSON formatting:

```
EXPLAIN (FORMAT JSON) SELECT * FROM names WHERE name = 'Joelle';
                  QUERY PLAN
-----------------------------------------------
 [                                             +
   {                                           +
     "Plan": {                                 +
       "Node Type": "Gather Motion",           +
       "Senders": 3,                           +
       "Receivers": 1,                         +
       "Slice": 1,                             +
       "Segments": 3,                          +
       "Gang Type": "primary reader",          +
       "Startup Cost": 0.00,                   +
       "Total Cost": 431.27,                   +
       "Plan Rows": 1,                         +
       "Plan Width": 58,                       +
       "Plans": [                              +
         {                                     +
           "Node Type": "Seq Scan",            +
           "Parent Relationship": "Outer",     +
           "Slice": 1,                         +
           "Segments": 3,                      +
           "Gang Type": "primary reader",      +
           "Relation Name": "names",           +
           "Alias": "names",                   +
           "Startup Cost": 0.00,               +
           "Total Cost": 431.27,               +
           "Plan Rows": 1,                     +
           "Plan Width": 58,                   +
           "Filter": "(name = 'Joelle'::text)"+
         }                                     +
       ]                                       +
     },                                        +
     "Settings": {                             +
       "Optimizer": "Pivotal Optimizer (GPORCA) version 3.23.0"      +
     }                                         +
   }                                           +
 ]
(1 row)
```

If there is an index and we use a query with an indexable `WHERE` condition, `EXPLAIN` might show a different plan. This query generates a plan with an index scan, with YAML formatting:

```
EXPLAIN (FORMAT YAML) SELECT * FROM NAMES WHERE LOCATION='Sydney,
 Australia';
                          QUERY PLAN
------------------------------------------------------------
 - Plan:                                                   +
     Node Type: "Gather Motion"                            +
     Senders: 3                                            +
     Receivers: 1                                          +
     Slice: 1                                              +
     Segments: 3                                           +
     Gang Type: "primary reader"                           +
     Startup Cost: 0.00                                    +
     Total Cost: 10.81                                     +
     Plan Rows: 10000                                      +
     Plan Width: 70                                        +
     Plans:                                                +
       - Node Type: "Index Scan"                           +
         Parent Relationship: "Outer"                      +
```

```
         Slice: 1                                                  +
         Segments: 3                                               +
         Gang Type: "primary reader"                               +
         Scan Direction: "Forward"                                 +
         Index Name: "names_idx_loc"                               +
         Relation Name: "names"                                    +
         Alias: "names"                                            +
         Startup Cost: 0.00                                        +
         Total Cost: 7.77                                          +
         Plan Rows: 10000                                          +
         Plan Width: 70                                            +
         Index Cond: "(location = 'Sydney, Australia'::text)"+
   Settings:                                                       +
     Optimizer: "Pivotal Optimizer (GPORCA) version 3.23.0"
(1 row)
```

Compatibility

There is no EXPLAIN statement defined in the SQL standard.

See Also

ANALYZE

FETCH

Retrieves rows from a query using a cursor.

Synopsis

```
FETCH [ forward_direction { FROM | IN } ] cursor_name
```

where *forward_direction* can be empty or one of:

```
    NEXT
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
```

Description

FETCH retrieves rows using a previously-created cursor.

> **Note:** This page describes usage of cursors at the SQL command level. If you are trying to use cursors inside a PL/pgSQL function, the rules are different. See *PL/pgSQL function*.

A cursor has an associated position, which is used by FETCH. The cursor position can be before the first row of the query result, on any particular row of the result, or after the last row of the result. When created, a cursor is positioned before the first row. After fetching some rows, the cursor is positioned on the row most recently retrieved. If FETCH runs off the end of the available rows then the cursor is left positioned after the last row. FETCH ALL will always leave the cursor positioned after the last row.

The forms NEXT, FIRST, LAST, ABSOLUTE, RELATIVE fetch a single row after moving the cursor appropriately. If there is no such row, an empty result is returned, and the cursor is left positioned before the first row or after the last row as appropriate.

The forms using FORWARD retrieve the indicated number of rows moving in the forward direction, leaving the cursor positioned on the last-returned row (or after all rows, if the count exceeds the number of rows available). Note that it is not possible to move a cursor position backwards in Greenplum Database, since scrollable cursors are not supported. You can only move a cursor forward in position using FETCH.

RELATIVE 0 and FORWARD 0 request fetching the current row without moving the cursor, that is, re-fetching the most recently fetched row. This will succeed unless the cursor is positioned before the first row or after the last row, in which case no row is returned.

Outputs

On successful completion, a FETCH command returns a command tag of the form

```
FETCH count
```

The count is the number of rows fetched (possibly zero). Note that in psql, the command tag will not actually be displayed, since psql displays the fetched rows instead.

Parameters

forward_direction

Defines the fetch direction and number of rows to fetch. Only forward fetches are allowed in Greenplum Database. It can be one of the following:

NEXT

Fetch the next row. This is the default if direction is omitted.

FIRST

Fetch the first row of the query (same as ABSOLUTE 1). Only allowed if it is the first FETCH operation using this cursor.

LAST

Fetch the last row of the query (same as ABSOLUTE -1).

ABSOLUTE *count*

Fetch the specified row of the query. Position after last row if count is out of range. Only allowed if the row specified by *count* moves the cursor position forward.

RELATIVE *count*

Fetch the specified row of the query *count* rows ahead of the current cursor position. RELATIVE 0 re-fetches the current row, if any. Only allowed if *count* moves the cursor position forward.

count

Fetch the next *count* number of rows (same as FORWARD *count*).

ALL

Fetch all remaining rows (same as FORWARD ALL).

FORWARD

Fetch the next row (same as NEXT).

FORWARD *count*

Fetch the next *count* number of rows. FORWARD 0 re-fetches the current row.

FORWARD ALL

Fetch all remaining rows.

cursor_name

The name of an open cursor.

Notes

Greenplum Database does not support scrollable cursors, so you can only use FETCH to move the cursor position forward.

ABSOLUTE fetches are not any faster than navigating to the desired row with a relative move: the underlying implementation must traverse all the intermediate rows anyway.

DECLARE is used to define a cursor. Use MOVE to change cursor position without retrieving data.

Examples

-- Start the transaction:

```
BEGIN;
```

-- Set up a cursor:

```
DECLARE mycursor CURSOR FOR SELECT * FROM films;
```

-- Fetch the first 5 rows in the cursor mycursor:

```
FETCH FORWARD 5 FROM mycursor;
 code  |          title          | did | date_prod  |   kind   |  len
-------+-------------------------+-----+------------+----------+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28
```

-- Close the cursor and end the transaction:

```
CLOSE mycursor;
COMMIT;
```

Change the kind column of the table films in the row at the c_films cursor's current position:

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

Compatibility

SQL standard allows cursors only in embedded SQL and in modules. Greenplum Database permits cursors to be used interactively.

The variant of FETCH described here returns the data as if it were a SELECT result rather than placing it in host variables. Other than this point, FETCH is fully upward-compatible with the SQL standard.

The FETCH forms involving FORWARD, as well as the forms FETCH count and FETCHALL, in which FORWARD is implicit, are Greenplum Database extensions. BACKWARD is not supported.

The SQL standard allows only FROM preceding the cursor name; the option to use IN, or to leave them out altogether, is an extension.

See Also

DECLARE, *CLOSE*, *MOVE*

GRANT

Defines access privileges.

Synopsis

```
GRANT { {SELECT | INSERT | UPDATE | DELETE | REFERENCES |
TRIGGER | TRUNCATE } [, ...] | ALL [PRIVILEGES] }
    ON { [TABLE] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO { [ GROUP ] role_name | PUBLIC} [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    TO { role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { {USAGE | SELECT | UPDATE} [, ...] | ALL [PRIVILEGES] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { {CREATE | CONNECT | TEMPORARY | TEMP} [, ...] | ALL
[PRIVILEGES] }
    ON DATABASE database_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [PRIVILEGES] }
    ON { FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...]
] ) [, ...]
         | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [PRIVILEGES] }
    ON LANGUAGE lang_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [, ...] | ALL [PRIVILEGES] }
    ON SCHEMA schema_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC}  [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [PRIVILEGES] }
    ON TABLESPACE tablespace_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT parent_role [, ...]
    TO member_role [, ...] [WITH ADMIN OPTION]

GRANT { SELECT | INSERT | ALL [PRIVILEGES] }
    ON PROTOCOL protocolname
    TO username
```

Description

Greenplum Database unifies the concepts of users and groups into a single kind of entity called a role. It is therefore not necessary to use the keyword GROUP to identify whether a grantee is a user or a group. GROUP is still allowed in the command, but it is a noise word.

The GRANT command has two basic variants: one that grants privileges on a database object (table, column, view, foreign table, sequence, database, foreign-data wrapper, foreign server, function, procedural language, schema, or tablespace), and one that grants membership in a role.

GRANT on Database Objects

This variant of the GRANT command gives specific privileges on a database object to one or more roles. These privileges are added to those already granted, if any.

There is also an option to grant privileges on all objects of the same type within one or more schemas. This functionality is currently supported only for tables, sequences, and functions (but note that ALL TABLES is considered to include views and foreign tables).

The keyword PUBLIC indicates that the privileges are to be granted to all roles, including those that may be created later. PUBLIC may be thought of as an implicitly defined group-level role that always includes all roles. Any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to PUBLIC.

If WITH GRANT OPTION is specified, the recipient of the privilege may in turn grant it to others. Without a grant option, the recipient cannot do that. Grant options cannot be granted to PUBLIC.

There is no need to grant privileges to the owner of an object (usually the role that created it), as the owner has all privileges by default. (The owner could, however, choose to revoke some of their own privileges for safety.)

The right to drop an object, or to alter its definition in any way is not treated as a grantable privilege; it is inherent in the owner, and cannot be granted or revoked. (However, a similar effect can be obtained by granting or revoking membership in the role that owns the object; see below.) The owner implicitly has all grant options for the object, too.

Greenplum Database grants default privileges on some types of objects to PUBLIC. No privileges are granted to PUBLIC by default on tables, table columns, sequences, foreign data wrappers, foreign servers, large objects, schemas, or tablespaces. For other types of objects, the default privileges granted to PUBLIC are as follows:

- CONNECT and TEMPORARY (create temporary tables) privileges for databases,
- EXECUTE privilege for functions, and
- USAGE privilege for languages and data types (including domains).

The object owner can, of course, REVOKE both default and expressly granted privileges. (For maximum security, issue the REVOKE in the same transaction that creates the object; then there is no window in which another user can use the object.)

>

GRANT on Roles

This variant of the GRANT command grants membership in a role to one or more other roles. Membership in a role is significant because it conveys the privileges granted to a role to each of its members.

If `WITH ADMIN OPTION` is specified, the member may in turn grant membership in the role to others, and revoke membership in the role as well. Without the admin option, ordinary users cannot do that. A role is not considered to hold `WITH ADMIN OPTION` on itself, but it may grant or revoke membership in itself from a database session where the session user matches the role. Database superusers can grant or revoke membership in any role to anyone. Roles having `CREATEROLE` privilege can grant or revoke membership in any role that is not a superuser.

Unlike the case with privileges, membership in a role cannot be granted to `PUBLIC`.

GRANT on Protocols

You can also use the `GRANT` command to specify which users can access a trusted protocol. (If the protocol is not trusted, you cannot give any other user permission to use it to read or write data.)

- To allow a user to create a readable external table with a trusted protocol:

```
GRANT SELECT ON PROTOCOL protocolname TO username
```

- To allow a user to create a writable external table with a trusted protocol:

```
GRANT INSERT ON PROTOCOL protocolname TO username
```

- To allow a user to create both readable and writable external table with a trusted protocol:

```
GRANT ALL ON PROTOCOL protocolname TO username
```

You can also use this command to grant users permissions to create and use `s3` and `pxf` external tables. However, external tables of type `http`, `https`, `gpfdist`, and `gpfdists`, are implemented internally in Greenplum Database instead of as custom protocols. For these types, use the `CREATE ROLE` or `ALTER ROLE` command to set the `CREATEEXTTABLE` or `NOCREATEEXTTABLE` attribute for each user. See *CREATE ROLE* for syntax and examples.

Parameters

SELECT

Allows `SELECT` from any column, or the specific columns listed, of the specified table, view, or sequence. Also allows the use of `COPY TO`. This privilege is also needed to reference existing column values in `UPDATE` or `DELETE`.

INSERT

Allows `INSERT` of a new row into the specified table. If specific columns are listed, only those columns may be assigned to in the `INSERT` command (other columns will receive default values). Also allows `COPY FROM`.

UPDATE

Allows `UPDATE` of any column, or the specific columns listed, of the specified table. `SELECT ... FOR UPDATE` and `SELECT ... FOR SHARE` also require this privilege on at least one column, (as well as the `SELECT` privilege). For sequences, this privilege allows the use of the `nextval()` and `setval()` functions.

DELETE

Allows `DELETE` of a row from the specified table.

REFERENCES

This keyword is accepted, although foreign key constraints are currently not supported in Greenplum Database. To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced columns. The privilege may be granted for all columns of a table, or just specific columns.

TRIGGER

Allows the creation of a trigger on the specified table.

Note: Greenplum Database does not support triggers.

TRUNCATE

Allows TRUNCATE of all rows from the specified table.

CREATE

For databases, allows new schemas to be created within the database.

For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object and have this privilege for the containing schema.

For tablespaces, allows tables and indexes to be created within the tablespace, and allows databases to be created that have the tablespace as their default tablespace. (Note that revoking this privilege will not alter the placement of existing objects.)

CONNECT

Allows the user to connect to the specified database. This privilege is checked at connection startup (in addition to checking any restrictions imposed by `pg_hba.conf`).

TEMPORARY
TEMP

Allows temporary tables to be created while using the database.

EXECUTE

Allows the use of the specified function and the use of any operators that are implemented on top of the function. This is the only type of privilege that is applicable to functions. (This syntax works for aggregate functions, as well.)

USAGE

For procedural languages, allows the use of the specified language for the creation of functions in that language. This is the only type of privilege that is applicable to procedural languages.

For schemas, allows access to objects contained in the specified schema (assuming that the objects' own privilege requirements are also met). Essentially this allows the grantee to look up objects within the schema.

For sequences, this privilege allows the use of the `currval()` and `nextval()` function.

For types and domains, this privilege allows the use of the type or domain in the creation of tables, functions, and other schema objects. (Note that it does not control general "usage" of the type, such as values of the type appearing in queries. It only prevents objects from being created that depend on the type. The main purpose of the privilege is controlling which users create dependencies on a type, which could prevent the owner from changing the type later.)

For foreign-data wrappers, this privilege enables the grantee to create new servers using that foreign-data wrapper.

For servers, this privilege enables the grantee to create foreign tables using the server, and also to create, alter, or drop their own user's user mappings associated with that server.

ALL PRIVILEGES

Grant all of the available privileges at once. The PRIVILEGES key word is optional in Greenplum Database, though it is required by strict SQL.

PUBLIC

A special group-level role that denotes that the privileges are to be granted to all roles, including those that may be created later.

WITH GRANT OPTION

The recipient of the privilege may in turn grant it to others.

WITH ADMIN OPTION

The member of a role may in turn grant membership in the role to others.

Notes

A user may perform SELECT, INSERT, and so forth, on a column if they hold that privilege for either the specific column or the whole table. Granting the privilege at the table level and then revoking it for one column does not do what you might wish: the table-level grant is unaffected by a column-level operation.

Database superusers can access all objects regardless of object privilege settings. One exception to this rule is view objects. Access to tables referenced in the view is determined by permissions of the view owner not the current user (even if the current user is a superuser).

If a superuser chooses to issue a GRANT or REVOKE command, the command is performed as though it were issued by the owner of the affected object. In particular, privileges granted via such a command will appear to have been granted by the object owner. For role membership, the membership appears to have been granted by the containing role itself.

GRANT and REVOKE can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges WITH GRANT OPTION on the object. In this case the privileges will be recorded as having been granted by the role that actually owns the object or holds the privileges WITH GRANT OPTION.

Granting permission on a table does not automatically extend permissions to any sequences used by the table, including sequences tied to SERIAL columns. Permissions on a sequence must be set separately.

The GRANT command cannot be used to set privileges for the protocols file, gpfdist, or gpfdists. These protocols are implemented internally in Greenplum Database. Instead, use the *CREATE ROLE* or *ALTER ROLE* command to set the CREATEEXTTABLE attribute for the role.

Use psql's \dp meta-command to obtain information about existing privileges for tables and columns. There are other \d meta-commands that you can use to display the privileges of non-table objects.

Examples

Grant insert privilege to all roles on table mytable:

```
GRANT INSERT ON mytable TO PUBLIC;
```

Grant all available privileges to role sally on the view topten. Note that while the above will indeed grant all privileges if executed by a superuser or the owner of topten, when executed by someone else it will only grant those permissions for which the granting role has grant options.

```
GRANT ALL PRIVILEGES ON topten TO sally;
```

Grant membership in role admins to user joe:

```
GRANT admins TO joe;
```

Compatibility

The PRIVILEGES key word is required in the SQL standard, but optional in Greenplum Database. The SQL standard does not support setting the privileges on more than one object per command.

Greenplum Database allows an object owner to revoke their own ordinary privileges: for example, a table owner can make the table read-only to theirself by revoking their own INSERT, UPDATE, DELETE, and TRUNCATE privileges. This is not possible according to the SQL standard. Greenplum Database treats the owner's privileges as having been granted by the owner to the owner; therefore they can revoke them too. In the SQL standard, the owner's privileges are granted by an assumed *system* entity.

The SQL standard provides for a USAGE privilege on other kinds of objects: character sets, collations, translations.

In the SQL standard, sequences only have a USAGE privilege, which controls the use of the NEXT VALUE FOR expression, which is equivalent to the function nextval in Greenplum Database. The sequence privileges SELECT and UPDATE are Greenplum Database extensions. The application of the sequence USAGE privilege to the currval function is also a Greenplum Database extension (as is the function itself).

Privileges on databases, tablespaces, schemas, and languages are Greenplum Database extensions.

See Also

REVOKE, *CREATE ROLE*, *ALTER ROLE*

INSERT

Creates new rows in a table.

Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table [( column [, ...] )]
   {DEFAULT VALUES | VALUES ( {expression | DEFAULT} [, ...] ) [, ...]
 | query}
   [RETURNING * | output_expression [[AS] output_name] [, ...]]
```

Description

INSERT inserts new rows into a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query.

The target column names may be listed in any order. If no list of column names is given at all, the default is the columns of the table in their declared order. The values supplied by the VALUES clause or query are associated with the explicit or implicit column list left-to-right.

Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or null if there is no default.

If the expression for any column is not of the correct data type, automatic type conversion will be attempted.

The optional RETURNING clause causes INSERT to compute and return value(s) based on each row actually inserted. This is primarily useful for obtaining values that were supplied by defaults, such as a serial sequence number. However, any expression using the table's columns is allowed. The syntax of the RETURNING list is identical to that of the output list of SELECT.

You must have INSERT privilege on a table in order to insert into it. When a column list is specified, you need INSERT privilege only on the listed columns. Use of the RETURNING clause requires SELECT privilege on all columns mentioned in RETURNING. If you provide a *query* to insert rows from a query, you must have SELECT privilege on any table or column referenced in the query.

Outputs

On successful completion, an INSERT command returns a command tag of the form:

```
INSERT oid count
```

The *count* is the number of rows inserted. If count is exactly one, and the target table has OIDs, then *oid* is the OID assigned to the inserted row. Otherwise *oid* is zero.

Parameters

with_query

The `WITH` clause allows you to specify one or more subqueries that can be referenced by name in the `INSERT` query.

For an `INSERT` command that includes a `WITH` clause, the clause can only contain `SELECT` statements, the `WITH` clause cannot contain a data-modifying command (`INSERT`, `UPDATE`, or `DELETE`).

It is possible for the query (`SELECT` statement) to also contain a `WITH` clause. In such a case both sets of *with_query* can be referenced within the `INSERT` query, but the second one takes precedence since it is more closely nested.

See *WITH Queries (Common Table Expressions)* and *SELECT* for details.

table

The name (optionally schema-qualified) of an existing table.

column

The name of a column in table. The column name can be qualified with a subfield name or array subscript, if needed. (Inserting into only some fields of a composite column leaves the other fields null.)

DEFAULT VALUES

All columns will be filled with their default values.

expression

An expression or value to assign to the corresponding column.

DEFAULT

The corresponding column will be filled with its default value.

query

A query (`SELECT` statement) that supplies the rows to be inserted. Refer to the `SELECT` statement for a description of the syntax.

output_expression

An expression to be computed and returned by the `INSERT` command after each row is inserted. The expression can use any column names of the table. Write * to return all columns of the inserted row(s).

output_name

A name to use for a returned column.

Notes

To insert data into a partitioned table, you specify the root partitioned table, the table created with the `CREATE TABLE` command. You also can specify a leaf child table of the partitioned table in an `INSERT` command. An error is returned if the data is not valid for the specified leaf child table. Specifying a child table that is not a leaf child table in the `INSERT` command is not supported. Execution of other DML commands such as `UPDATE` and `DELETE` on any child table of a partitioned table is not supported. These commands must be executed on the root partitioned table, the table created with the `CREATE TABLE` command.

For a partitioned table, all the child tables are locked during the `INSERT` operation when the Global Deadlock Detector is not enabled (the default). Only some of the leaf child tables are locked when the Global Deadlock Detector is enabled. For information about the Global Deadlock Detector, see *Global Deadlock Detector*.

For append-optimized tables, Greenplum Database supports a maximum of 127 concurrent `INSERT` transactions into a single append-optimized table.

For writable S3 external tables, the `INSERT` operation uploads to one or more files in the configured S3 bucket, as described in *s3:// Protocol*. Pressing `Ctrl-c` cancels the `INSERT` and stops uploading to S3.

Examples

Insert a single row into table `films`:

```
INSERT INTO films VALUES ('UA502', 'Bananas', 105,
'1971-07-13', 'Comedy', '82 minutes');
```

In this example, the `length` column is omitted and therefore it will have the default value:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

This example uses the `DEFAULT` clause for the `date_prod` column rather than specifying a value:

```
INSERT INTO films VALUES ('UA502', 'Bananas', 105, DEFAULT,
'Comedy', '82 minutes');
```

To insert a row consisting entirely of default values:

```
INSERT INTO films DEFAULT VALUES;
```

To insert multiple rows using the multirow `VALUES` syntax:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

This example inserts some rows into table `films` from a table `tmp_films` with the same column layout as `films`:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod <
'2004-05-07';
```

Insert a single row into table distributors, returning the sequence number generated by the *DEFAULT* clause:

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
   RETURNING did;
```

Compatibility

`INSERT` conforms to the SQL standard. The case in which a column name list is omitted, but not all the columns are filled from the `VALUES` clause or query, is disallowed by the standard.

Possible limitations of the *query* clause are documented under `SELECT`.

See Also

`COPY`, *`SELECT`*, *`CREATE EXTERNAL TABLE`*, *s3:// Protocol*

LOAD

Loads or reloads a shared library file.

Synopsis

```
LOAD 'filename'
```

Description

This command loads a shared library file into the Greenplum Database server address space. If the file had been loaded previously, it is first unloaded. This command is primarily useful to unload and reload a shared library file that has been changed since the server first loaded it. To make use of the shared library, function(s) in it need to be declared using the `CREATE FUNCTION` command.

The file name is specified in the same way as for shared library names in `CREATE FUNCTION`; in particular, one may rely on a search path and automatic addition of the system's standard shared library file name extension.

Note that in Greenplum Database the shared library file (`.so` file) must reside in the same path location on every host in the Greenplum Database array (masters, segments, and mirrors).

Only database superusers can load shared library files.

Parameters

filename

The path and file name of a shared library file. This file must exist in the same location on all hosts in your Greenplum Database array.

Examples

Load a shared library file:

```
LOAD '/usr/local/greenplum-db/lib/myfuncs.so';
```

Compatibility

`LOAD` is a Greenplum Database extension.

See Also

CREATE FUNCTION

LOCK

Locks a table.

Synopsis

```
LOCK [TABLE] [ONLY] name [ * ] [, ...] [IN lockmode MODE] [NOWAIT]
```

where *lockmode* is one of:

```
    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
  | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Description

`LOCK TABLE` obtains a table-level lock, waiting if necessary for any conflicting locks to be released. If `NOWAIT` is specified, `LOCK TABLE` does not wait to acquire the desired lock: if it cannot be acquired immediately, the command is aborted and an error is emitted. Once obtained, the lock is held for the

remainder of the current transaction. There is no UNLOCK TABLE command; locks are always released at transaction end.

When acquiring locks automatically for commands that reference tables, Greenplum Database always uses the least restrictive lock mode possible. LOCK TABLE provides for cases when you might need more restrictive locking. For example, suppose an application runs a transaction at the *Read Committed* isolation level and needs to ensure that data in a table remains stable for the duration of the transaction. To achieve this you could obtain SHARE lock mode over the table before querying. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because SHARE lock mode conflicts with the ROW EXCLUSIVE lock acquired by writers, and your LOCK TABLE *name* IN SHARE MODE statement will wait until any concurrent holders of ROW EXCLUSIVE mode locks commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes outstanding; furthermore none can begin until you release the lock.

To achieve a similar effect when running a transaction at the REPEATABLE READ or SERIALIZABLE isolation level, you have to execute the LOCK TABLE statement before executing any SELECT or data modification statement. A REPEATABLE READ or SERIALIZABLE transaction's view of data will be frozen when its first SELECT or data modification statement begins. A LOCK TABLE later in the transaction will still prevent concurrent writes — but it won't ensure that what the transaction reads corresponds to the latest committed values.

If a transaction of this sort is going to change the data in the table, then it should use SHARE ROW EXCLUSIVE lock mode instead of SHARE mode. This ensures that only one transaction of this type runs at a time. Without this, a deadlock is possible: two transactions might both acquire SHARE mode, and then be unable to also acquire ROW EXCLUSIVE mode to actually perform their updates. Note that a transaction's own locks never conflict, so a transaction can acquire ROW EXCLUSIVE mode when it holds SHARE mode — but not if anyone else holds SHARE mode. To avoid deadlocks, make sure all transactions acquire locks on the same objects in the same order, and if multiple lock modes are involved for a single object, then transactions should always acquire the most restrictive mode first.

Parameters

name

The name (optionally schema-qualified) of an existing table to lock. If ONLY is specified, only that table is locked. If ONLY is not specified, the table and all its descendant tables (if any) are locked. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

If multiple tables are given, tables are locked one-by-one in the order specified in the LOCK TABLE command.

lockmode

The lock mode specifies which locks this lock conflicts with. If no lock mode is specified, then ACCESS EXCLUSIVE, the most restrictive mode, is used. Lock modes are as follows:

- ACCESS SHARE — Conflicts with the ACCESS EXCLUSIVE lock mode only. The SELECT command acquires a lock of this mode on referenced tables. In general, any query that only reads a table and does not modify it will acquire this lock mode.
- ROW SHARE — Conflicts with the EXCLUSIVE and ACCESS EXCLUSIVE lock modes. The SELECT FOR SHARE command automatically acquires a lock of this mode on the target table(s) (in addition to ACCESS SHARE locks on any other tables that are referenced but not selected FOR SHARE).
- ROW EXCLUSIVE — Conflicts with the SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. The commands INSERT and COPY automatically acquire this lock mode on the target table (in addition to ACCESS SHARE locks on any other referenced tables) See *Note*.
- SHARE UPDATE EXCLUSIVE — Conflicts with the SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.

This mode protects a table against concurrent schema changes and VACUUM runs. Acquired by VACUUM (without FULL) on heap tables and ANALYZE.

- SHARE — Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent data changes. Acquired automatically by CREATE INDEX.
- SHARE ROW EXCLUSIVE — Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This lock mode is not automatically acquired by any Greenplum Database command.
- EXCLUSIVE — Conflicts with the ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode allows only concurrent ACCESS SHARE locks, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode. This lock mode is automatically acquired for UPDATE, SELECT FOR UPDATE, and DELETE in Greenplum Database (which is more restrictive locking than in regular PostgreSQL). See *Note*.
- ACCESS EXCLUSIVE — Conflicts with locks of all modes (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHAREROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE). This mode guarantees that the holder is the only transaction accessing the table in any way. Acquired automatically by the ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, and VACUUM FULL commands. This is the default lock mode for LOCK TABLE statements that do not specify a mode explicitly. This lock is also briefly acquired by VACUUM (without FULL) on append-optimized tables during processing.

  Note: By default Greenplum Database acquires the more restrictive EXCLUSIVE lock (rather than ROW EXCLUSIVE in PostgreSQL) for UPDATE, DELETE, and SELECT...FOR UPDATE operations on heap tables. When the Global Deadlock Detector is enabled the lock mode for UPDATE and DELETE operations on heap tables is ROW EXCLUSIVE. See *Global Deadlock Detector*. Greenplum always holds a table-level lock with SELECT...FOR UPDATE statements.

NOWAIT

Specifies that LOCK TABLE should not wait for any conflicting locks to be released: if the specified lock(s) cannot be acquired immediately without waiting, the transaction is aborted.

Notes

LOCK TABLE ... IN ACCESS SHARE MODE requires SELECT privileges on the target table. All other forms of LOCK require table-level UPDATE, DELETE, or TRUNCATE privileges.

LOCK TABLE is useless outside of a transaction block: the lock would be held only to the completion of the LOCK statement. Therefore, Greenplum Database reports an error if LOCK is used outside of a transaction block. Use BEGIN and END to define a transaction block.

LOCK TABLE only deals with table-level locks, and so the mode names involving ROW are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, ROW EXCLUSIVE mode is a shareable table lock. Keep in mind that all the lock modes have identical semantics so far as LOCK TABLE is concerned, differing only in the rules about which modes conflict with which. For information on how to acquire an actual row-level lock, see the FOR UPDATE/FOR SHARE clause in the *SELECT* reference documentation.

Examples

Obtain a SHARE lock on the `films` table when going to perform inserts into the `films_user_comments` table:

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- Do ROLLBACK if record was not returned
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

Take a `SHARE ROW EXCLUSIVE` lock on a table when performing a delete operation:

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

Compatibility

There is no `LOCK TABLE` in the SQL standard, which instead uses `SET TRANSACTION` to specify concurrency levels on transactions. Greenplum Database supports that too.

Except for `ACCESS SHARE`, `ACCESS EXCLUSIVE`, and `SHARE UPDATE EXCLUSIVE` lock modes, the Greenplum Database lock modes and the `LOCK TABLE` syntax are compatible with those present in Oracle.

See Also

`BEGIN`, *`SET TRANSACTION`*, *`SELECT`*

MOVE

Positions a cursor.

Synopsis

```
MOVE [ forward_direction [ FROM | IN ] ] cursor_name
```

where *forward_direction* can be empty or one of:

```
    NEXT
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
```

Description

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the *FETCH* command, except it only positions the cursor and does not return rows.

Note that it is not possible to move a cursor position backwards in Greenplum Database, since scrollable cursors are not supported. You can only move a cursor forward in position using MOVE.

Outputs

On successful completion, a MOVE command returns a command tag of the form

```
MOVE count
```

The count is the number of rows that a FETCH command with the same parameters would have returned (possibly zero).

Parameters

forward_direction

The parameters for the MOVE command are identical to those of the FETCH command; refer to *FETCH* for details on syntax and usage.

cursor_name

The name of an open cursor.

Examples

-- Start the transaction:

```
BEGIN;
```

-- Set up a cursor:

```
DECLARE mycursor CURSOR FOR SELECT * FROM films;
```

-- Move forward 5 rows in the cursor mycursor:

```
MOVE FORWARD 5 IN mycursor;
MOVE 5
```

--Fetch the next row after that (row 6):

```
FETCH 1 FROM mycursor;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)
```

-- Close the cursor and end the transaction:

```
CLOSE mycursor;
COMMIT;
```

Compatibility

There is no MOVE statement in the SQL standard.

See Also

DECLARE, *FETCH*, *CLOSE*

PREPARE

Prepare a statement for execution.

Synopsis

```
PREPARE name [ (datatype [, ...] ) ] AS statement
```

Description

PREPARE creates a prepared statement. A prepared statement is a server-side object that can be used to optimize performance. When the PREPARE statement is executed, the specified statement is parsed, analyzed, and rewritten. When an EXECUTE command is subsequently issued, the prepared statement is planned and executed. This division of labor avoids repetitive parse analysis work, while allowing the execution plan to depend on the specific parameter values supplied.

Prepared statements can take parameters, values that are substituted into the statement when it is executed. When creating the prepared statement, refer to parameters by position, using $1, $2, etc. A corresponding list of parameter data types can optionally be specified. When a parameter's data type is not specified or is declared as unknown, the type is inferred from the context in which the parameter is first used (if possible). When executing the statement, specify the actual values for these parameters in the EXECUTE statement.

Prepared statements only last for the duration of the current database session. When the session ends, the prepared statement is forgotten, so it must be recreated before being used again. This also means that a single prepared statement cannot be used by multiple simultaneous database clients; however, each client can create their own prepared statement to use. Prepared statements can be manually cleaned up using the *DEALLOCATE* command.

Prepared statements have the largest performance advantage when a single session is being used to execute a large number of similar statements. The performance difference will be particularly significant if the statements are complex to plan or rewrite, for example, if the query involves a join of many tables or requires the application of several rules. If the statement is relatively simple to plan and rewrite but relatively expensive to execute, the performance advantage of prepared statements will be less noticeable.

Parameters

name

An arbitrary name given to this particular prepared statement. It must be unique within a single session and is subsequently used to execute or deallocate a previously prepared statement.

datatype

The data type of a parameter to the prepared statement. If the data type of a particular parameter is unspecified or is specified as unknown, it will be inferred from the context in which the parameter is first used. To refer to the parameters in the prepared statement itself, use $1, $2, etc.

statement

Any SELECT, INSERT, UPDATE, DELETE, or VALUES statement.

Notes

A prepared statement can be executed with either a *generic plan* or a *custom plan*. A generic plan is the same across all executions, while a custom plan is generated for a specific execution using the parameter values given in that call. Use of a generic plan avoids planning overhead, but in some situations a custom

plan will be much more efficient to execute because the planner can make use of knowledge of the parameter values. If the prepared statement has no parameters, a generic plan is always used.

By default (with the default value, `auto`, for the server configuration parameter *`plan_cache_mode`*), the server automatically chooses whether to use a generic or custom plan for a prepared statement that has parameters. The current rule for this is that the first five executions are done with custom plans and the average estimated cost of those plans is calculated. Then a generic plan is created and its estimated cost is compared to the average custom-plan cost. Subsequent executions use the generic plan if its cost is not so much higher than the average custom-plan cost as to make repeated replanning seem preferable.

This heuristic can be overridden, forcing the server to use either generic or custom plans, by setting `plan_cache_mode` to `force_generic_plan` or `force_custom_plan` respectively. This setting is primarily useful if the generic plan's cost estimate is badly off for some reason, allowing it to be chosen even though its actual cost is much more than that of a custom plan.

To examine the query plan Greenplum Database is using for a prepared statement, use *`EXPLAIN`*, for example

```
EXPLAIN EXECUTE <name>(<parameter_values>);
```

If a generic plan is in use, it will contain parameter symbols $n, while a custom plan will have the supplied parameter values substituted into it.

For more information on query planning and the statistics collected by Greenplum Database for that purpose, see the ANALYZE documentation.

Although the main point of a prepared statement is to avoid repeated parse analysis and planning of the statement, Greenplum will force re-analysis and re-planning of the statement before using it whenever database objects used in the statement have undergone definitional (DDL) changes since the previous use of the prepared statement. Also, if the value of `search_path` changes from one use to the next, the statement will be re-parsed using the new search_path. (This latter behavior is new as of Greenplum 6.) These rules make use of a prepared statement semantically almost equivalent to re-submitting the same query text over and over, but with a performance benefit if no object definitions are changed, especially if the best plan remains the same across uses. An example of a case where the semantic equivalence is not perfect is that if the statement refers to a table by an unqualified name, and then a new table of the same name is created in a schema appearing earlier in the `search_path`, no automatic re-parse will occur since no object used in the statement changed. However, if some other change forces a re-parse, the new table will be referenced in subsequent uses.

You can see all prepared statements available in the session by querying the `pg_prepared_statements` system view.

Examples

Create a prepared statement for an `INSERT` statement, and then execute it:

```
PREPARE fooplan (int, text, bool, numeric) AS INSERT INTO
foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Create a prepared statement for a `SELECT` statement, and then execute it. Note that the data type of the second parameter is not specified, so it is inferred from the context in which `$2` is used:

```
PREPARE usrrptplan (int) AS SELECT * FROM users u, logs l
WHERE u.usrid=$1 AND u.usrid=l.usrid AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

Compatibility

The SQL standard includes a PREPARE statement, but it can only be used in embedded SQL, and it uses a different syntax.

See Also

EXECUTE, *DEALLOCATE*

REASSIGN OWNED

Changes the ownership of database objects owned by a database role.

Synopsis

```
REASSIGN OWNED BY old_role [, ...] TO new_role
```

Description

REASSIGN OWNED changes the ownership of database objects owned by any of the *old_role*s to *new_role*.

Parameters

old_role

The name of a role. The ownership of all the objects in the current database, and of all shared objects (databases, tablespaces), owned by this role will be reassigned to *new_role*.

new_role

The name of the role that will be made the new owner of the affected objects.

Notes

REASSIGN OWNED is often used to prepare for the removal of one or more roles. Because REASSIGN OWNED does not affect objects in other databases, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed.

REASSIGN OWNED requires privileges on both the source role(s) and the target role.

The *DROP OWNED* command is an alternative that simply drops all of the database objects owned by one or more roles. DROP OWNED requires privileges only on the source role(s).

The REASSIGN OWNED command does not affect any privileges granted to the *old_role*s on objects that are not owned by them. Likewise, it does not affect default privileges created with ALTER DEFAULT PRIVILEGES. Use DROP OWNED to revoke such privileges.

Examples

Reassign any database objects owned by the role named sally and bob to admin;

```
REASSIGN OWNED BY sally, bob TO admin;
```

Compatibility

The REASSIGN OWNED command is a Greenplum Database extension.

See Also

DROP OWNED, *DROP ROLE*, *ALTER DATABASE*

REFRESH MATERIALIZED VIEW

Replaces the contents of a materialized view.

Synopsis

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name
    [ WITH [ NO ] DATA ]
```

Description

`REFRESH MATERIALIZED VIEW` completely replaces the contents of a materialized view. The old contents are discarded. To execute this command you must be the owner of the materialized view. With the default, `WITH DATA`, the materialized view query is executed to provide the new data, and the materialized view is left in a scannable state. If `WITH NO DATA` is specified, no new data is generated and the materialized view is left in an unscannable state. A query returns an error if the query attempts to access the materialized view.

Parameters

`CONCURRENTLY`

Refresh the materialized view without locking out concurrent selects on the materialized view. Without this option, a refresh that affects a lot of rows tends to use fewer resources and completes more quickly, but could block other connections which are trying to read from the materialized view. This option might be faster in cases where a small number of rows are affected.

This option is only allowed if there is at least one `UNIQUE` index on the materialized view which uses only column names and includes all rows; that is, it must not index on any expressions nor include a `WHERE` clause.

This option cannot be used when the materialized view is not already populated, and it cannot be used with the `WITH NO DATA` clause.

Even with this option, only one `REFRESH` at a time may run against any one materialized view.

`name`

The name (optionally schema-qualified) of the materialized view to refresh.

`WITH [ NO ] DATA`

`WITH DATA` is the default and specifies that the materialized view query is executed to provide new data, and the materialized view is left in a scannable state. If `WITH NO DATA` is specified, no new data is generated and the materialized view is left in an unscannable state. An error is returned if a query attempts to access an unscannable materialized view.

`WITH NO DATA` cannot be used with `CONCURRENLY`.

Notes

While the default index for future *`CLUSTER`* operations is retained, `REFRESH MATERIALIZED VIEW` does not order the generated rows based on this property. If you want the data to be ordered upon generation, you must use an `ORDER BY` clause in the materialized view query. However, if a materialized view query contains an `ORDER BY` or `SORT` clause, the data is not guaranteed to be ordered or sorted if `SELECT` is performed on the materialized view.

Examples

This command replaces the contents of the materialized view `order_summary` using the query from the materialized view's definition, and leaves it in a scannable state.

```
REFRESH MATERIALIZED VIEW order_summary;
```

This command frees storage associated with the materialized view `annual_statistics_basis` and leaves it in an unscannable state.

```
REFRESH MATERIALIZED VIEW annual_statistics_basis WITH NO DATA;
```

Compatibility

`REFRESH MATERIALIZED VIEW` is a Greenplum Database extension of the SQL standard.

See Also

ALTER MATERIALIZED VIEW, *CREATE MATERIALIZED VIEW*, *DROP MATERIALIZED VIEW*

REINDEX

Rebuilds indexes.

Synopsis

```
REINDEX {INDEX | TABLE | DATABASE | SYSTEM} name
```

Description

`REINDEX` rebuilds an index using the data stored in the index's table, replacing the old copy of the index. There are several scenarios in which to use `REINDEX`:

- An index has become bloated, that is, it contains many empty or nearly-empty pages. This can occur with B-tree indexes in Greenplum Database under certain uncommon access patterns. `REINDEX` provides a way to reduce the space consumption of the index by writing a new version of the index without the dead pages.
- You have altered the `FILLFACTOR` storage parameter for an index, and wish to ensure that the change has taken full effect.

Parameters

`INDEX`

Recreate the specified index.

`TABLE`

Recreate all indexes of the specified table. If the table has a secondary TOAST table, that is reindexed as well.

`DATABASE`

Recreate all indexes within the current database. Indexes on shared system catalogs are also processed. This form of `REINDEX` cannot be executed inside a transaction block.

`SYSTEM`

Recreate all indexes on system catalogs within the current database. Indexes on shared system catalogs are included. Indexes on user tables are not processed. This form of `REINDEX` cannot be executed inside a transaction block.

name

The name of the specific index, table, or database to be reindexed. Index and table names may be schema-qualified. Presently, `REINDEX DATABASE` and `REINDEX SYSTEM` can only reindex the current database, so their parameter must match the current database's name.

Notes

`REINDEX` is similar to a drop and recreate of the index in that the index contents are rebuilt from scratch. However, the locking considerations are rather different. `REINDEX` locks out writes but not reads of the index's parent table. It also takes an exclusive lock on the specific index being processed, which will block reads that attempt to use that index. In contrast, `DROP INDEX` momentarily takes an exclusive lock on the parent table, blocking both writes and reads. The subsequent `CREATE INDEX` locks out writes but not reads; since the index is not there, no read will attempt to use it, meaning that there will be no blocking but reads may be forced into expensive sequential scans.

Reindexing a single index or table requires being the owner of that index or table. Reindexing a database requires being the owner of the database (note that the owner can therefore rebuild indexes of tables owned by other users). Of course, superusers can always reindex anything.

`REINDEX` does not update the `reltuples` and `relpages` statistics for the index. To update those statistics, run `ANALYZE` on the table after reindexing.

If you suspect that shared global system catalog indexes are corrupted, they can only be reindexed in Greenplum utility mode. The typical symptom of a corrupt shared index is "index is not a btree" errors, or else the server crashes immediately at startup due to reliance on the corrupted indexes. Contact Greenplum Customer Support for assistance in this situation.

Examples

Rebuild a single index:

```
REINDEX INDEX my_index;
```

Rebuild all the indexes on the table `my_table`:

```
REINDEX TABLE my_table;
```

Compatibility

There is no `REINDEX` command in the SQL standard.

See Also

`CREATE INDEX`, *`DROP INDEX`*, *`VACUUM`*

RELEASE SAVEPOINT

Destroys a previously defined savepoint.

Synopsis

```
RELEASE [SAVEPOINT] savepoint_name
```

Description

`RELEASE SAVEPOINT` destroys a savepoint previously defined in the current transaction.

Destroying a savepoint makes it unavailable as a rollback point, but it has no other user visible behavior. It does not undo the effects of commands executed after the savepoint was established. (To do that, see *ROLLBACK TO SAVEPOINT*.) Destroying a savepoint when it is no longer needed may allow the system to reclaim some resources earlier than transaction end.

`RELEASE SAVEPOINT` also destroys all savepoints that were established *after* the named savepoint was established.

Parameters

savepoint_name

The name of the savepoint to destroy.

Examples

To establish and later destroy a savepoint:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

The above transaction will insert both 3 and 4.

Compatibility

This command conforms to the SQL standard. The standard specifies that the key word `SAVEPOINT` is mandatory, but Greenplum Database allows it to be omitted.

See Also

BEGIN, *SAVEPOINT*, *ROLLBACK TO SAVEPOINT*, *COMMIT*

RESET

Restores the value of a system configuration parameter to the default value.

Synopsis

```
RESET configuration_parameter

RESET ALL
```

Description

`RESET` restores system configuration parameters to their default values. `RESET` is an alternative spelling for `SET` *`configuration_parameter`* `TO DEFAULT`.

The default value is defined as the value that the parameter would have had, had no `SET` ever been issued for it in the current session. The actual source of this value might be a compiled-in default, the master `postgresql.conf` configuration file, command-line options, or per-database or per-user default settings. See *Server Configuration Parameters* for more information.

Parameters

configuration_parameter

The name of a system configuration parameter. See *Server Configuration Parameters* for details.

ALL

Resets all settable configuration parameters to their default values.

Examples

Set the `statement_mem` configuration parameter to its default value:

```
RESET statement_mem;
```

Compatibility

RESET is a Greenplum Database extension.

See Also

SET

REVOKE

Removes access privileges.

Synopsis

```
REVOKE [GRANT OPTION FOR] { {SELECT | INSERT | UPDATE | DELETE
       | REFERENCES | TRIGGER | TRUNCATE } [, ...] | ALL [PRIVILEGES] }

       ON { [TABLE] table_name [, ...]
            | ALL TABLES IN SCHEMA schema_name [, ...] }
       FROM { [ GROUP ] role_name | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [ GRANT OPTION FOR ] { { SELECT | INSERT | UPDATE
       | REFERENCES } ( column_name [, ...] )
       [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
       ON [ TABLE ] table_name [, ...]
       FROM { [ GROUP ]  role_name | PUBLIC } [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [GRANT OPTION FOR] { {USAGE | SELECT | UPDATE} [,...]
       | ALL [PRIVILEGES] }
       ON { SEQUENCE sequence_name [, ...]
            | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {CREATE | CONNECT
       | TEMPORARY | TEMP} [, ...] | ALL [PRIVILEGES] }
       ON DATABASE database_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [ GRANT OPTION FOR ]
       { USAGE | ALL [ PRIVILEGES ] }
       ON DOMAIN domain_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
       { USAGE | ALL [ PRIVILEGES ] }
       ON FOREIGN DATA WRAPPER fdw_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
       { USAGE | ALL [ PRIVILEGES ] }
       ON FOREIGN SERVER server_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [GRANT OPTION FOR] {EXECUTE | ALL [PRIVILEGES]}
       ON { FUNCTION funcname ( [[argmode] [argname] argtype
                                [, ...]] ) [, ...]
            | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
       FROM { [ GROUP ] role_name | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {USAGE | ALL [PRIVILEGES]}
       ON LANGUAGE langname [, ...]
       FROM { [ GROUP ]  role_name | PUBLIC} [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [GRANT OPTION FOR] { {CREATE | USAGE} [, ...]
       | ALL [PRIVILEGES] }
       ON SCHEMA schema_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { CREATE | ALL [PRIVILEGES] }
       ON TABLESPACE tablespacename [, ...]
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [ GRANT OPTION FOR ]
       { USAGE | ALL [ PRIVILEGES ] }
       ON TYPE type_name [, ...]
       FROM { [ GROUP ] role_name | PUBLIC } [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [ADMIN OPTION FOR] parent_role [, ...]
       FROM [ GROUP ] member_role [, ...]
       [CASCADE | RESTRICT]
```

Description

REVOKE command revokes previously granted privileges from one or more roles. The key word PUBLIC refers to the implicitly defined group of all roles.

See the description of the *GRANT* command for the meaning of the privilege types.

Note that any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to PUBLIC. Thus, for example, revoking SELECT privilege from PUBLIC does not necessarily mean that all roles have lost SELECT privilege on the object: those who have it granted directly or via another role will still have it. Similarly, revoking SELECT from a user might not prevent that user from using SELECT if PUBLIC or another membership role still has SELECT rights.

If GRANT OPTION FOR is specified, only the grant option for the privilege is revoked, not the privilege itself. Otherwise, both the privilege and the grant option are revoked.

If a role holds a privilege with grant option and has granted it to other roles then the privileges held by those other roles are called dependent privileges. If the privilege or the grant option held by the first role is being revoked and dependent privileges exist, those dependent privileges are also revoked if CASCADE is specified, else the revoke action will fail. This recursive revocation only affects privileges that were granted through a chain of roles that is traceable to the role that is the subject of this REVOKE command. Thus, the affected roles may effectively keep the privilege if it was also granted through other roles.

When you revoke privileges on a table, Greenplum Database revokes the corresponding column privileges (if any) on each column of the table, as well. On the other hand, if a role has been granted privileges on a table, then revoking the same privileges from individual columns will have no effect.

When revoking membership in a role, GRANT OPTION is instead called ADMIN OPTION, but the behavior is similar.

Parameters

See *GRANT*.

Notes

A user may revoke only those privileges directly granted by that user. If, for example, user A grants a privilege with grant option to user B, and user B has in turn granted it to user C, then user A cannot revoke the privilege directly from C. Instead, user A could revoke the grant option from user B and use the CASCADE option so that the privilege is in turn revoked from user C. For another example, if both A and B grant the same privilege to C, A can revoke his own grant but not B's grant, so C effectively still has the privilege.

When a non-owner of an object attempts to REVOKE privileges on the object, the command fails outright if the user has no privileges whatsoever on the object. As long as some privilege is available, the command proceeds, but it will revoke only those privileges for which the user has grant options. The REVOKE ALL PRIVILEGES forms issue a warning message if no grant options are held, while the other forms issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since Greenplum Database always treats the owner as holding all grant options, the cases can never occur.)

If a superuser chooses to issue a GRANT or REVOKE command, Greenplum Database performs the command as though it were issued by the owner of the affected object. Since all privileges ultimately come from the object owner (possibly indirectly via chains of grant options), it is possible for a superuser to revoke all privileges, but this might require use of CASCADE as stated above.

REVOKE may also be invoked by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges WITH GRANT OPTION on the object. In this case, Greenplum Database performs the command as though it were issued by the containing role that actually owns the object or holds the privileges WITH GRANT OPTION. For example, if table t1 is owned by role g1, of which role u1 is a member, then u1 can revoke privileges on t1 that are recorded as being granted by g1. This includes grants made by u1 as well as by other members of role g1.

If the role that executes REVOKE holds privileges indirectly via more than one role membership path, it is unspecified which containing role will be used to perform the command. In such cases it is best practice to use SET ROLE to become the specific role as which you want to do the REVOKE. Failure to do so may lead to revoking privileges other than the ones you intended, or not revoking any privileges at all.

Use psql's \dp meta-command to obtain information about existing privileges for tables and columns. There are other \d meta-commands that you can use to display the privileges of non-table objects.

Examples

Revoke insert privilege for the public on table films:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Revoke all privileges from role `sally` on view `topten`. Note that this actually means revoke all privileges that the current role granted (if not a superuser).

```
REVOKE ALL PRIVILEGES ON topten FROM sally;
```

Revoke membership in role `admins` from user `joe`:

```
REVOKE admins FROM joe;
```

Compatibility

The compatibility notes of the *GRANT* command also apply to `REVOKE`.

Either `RESTRICT` or `CASCADE` is required according to the standard, but Greenplum Database assumes `RESTRICT` by default.

See Also

GRANT

ROLLBACK

Aborts the current transaction.

Synopsis

```
ROLLBACK [WORK | TRANSACTION]
```

Description

`ROLLBACK` rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Parameters

`WORK`
`TRANSACTION`

Optional key words. They have no effect.

Notes

Use `COMMIT` to successfully end the current transaction.

Issuing `ROLLBACK` when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To discard all changes made in the current transaction:

```
ROLLBACK;
```

Compatibility

The SQL standard only specifies the two forms `ROLLBACK` and `ROLLBACK WORK`. Otherwise, this command is fully conforming.

See Also

BEGIN, *COMMIT*, *SAVEPOINT*, *ROLLBACK TO SAVEPOINT*

ROLLBACK TO SAVEPOINT

Rolls back the current transaction to a savepoint.

Synopsis

```
ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] savepoint_name
```

Description

This command will roll back all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

`ROLLBACK TO SAVEPOINT` implicitly destroys all savepoints that were established after the named savepoint.

Parameters

`WORK`
`TRANSACTION`

Optional key words. They have no effect.

`savepoint_name`

The name of a savepoint to roll back to.

Notes

Use `RELEASE SAVEPOINT` to destroy a savepoint without discarding the effects of commands executed after it was established.

Specifying a savepoint name that has not been established is an error.

Cursors have somewhat non-transactional behavior with respect to savepoints. Any cursor that is opened inside a savepoint will be closed when the savepoint is rolled back. If a previously opened cursor is affected by a `FETCH` command inside a savepoint that is later rolled back, the cursor remains at the position that `FETCH` left it pointing to (that is, cursor motion caused by `FETCH` is not rolled back). Closing a cursor is not undone by rolling back, either. However, other side-effects caused by the cursor's query (such as side-effects of volatile functions called by the query) are rolled back if they occur during a savepoint that is later rolled back. A cursor whose execution causes a transaction to abort is put in a cannot-execute state, so while the transaction can be restored using `ROLLBACK TO SAVEPOINT`, the cursor can no longer be used.

Examples

To undo the effects of the commands executed after `my_savepoint` was established:

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Cursor positions are not affected by a savepoint rollback:

```
BEGIN;
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;
SAVEPOINT foo;
FETCH 1 FROM foo;
column
----------
```

```
        1
ROLLBACK TO SAVEPOINT foo;
FETCH 1 FROM foo;
column
----------
        2
COMMIT;
```

Compatibility

The SQL standard specifies that the key word SAVEPOINT is mandatory, but Greenplum Database (and Oracle) allow it to be omitted. SQL allows only WORK, not TRANSACTION, as a noise word after ROLLBACK. Also, SQL has an optional clause AND [NO] CHAIN which is not currently supported by Greenplum Database. Otherwise, this command conforms to the SQL standard.

See Also

BEGIN, *COMMIT*, *SAVEPOINT*, *RELEASE SAVEPOINT*, *ROLLBACK*

SAVEPOINT

Defines a new savepoint within the current transaction.

Synopsis

```
SAVEPOINT savepoint_name
```

Description

SAVEPOINT establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

Parameters

savepoint_name

The name of the new savepoint.

Notes

Use *ROLLBACK TO SAVEPOINT* to rollback to a savepoint. Use *RELEASE SAVEPOINT* to destroy a savepoint, keeping the effects of commands executed after it was established.

Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.

Examples

To establish a savepoint and later undo the effects of all commands executed after it was established:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

The above transaction will insert the values 1 and 3, but not 2.

To establish and later destroy a savepoint:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

The above transaction will insert both 3 and 4.

Compatibility

SQL requires a savepoint to be destroyed automatically when another savepoint with the same name is established. In Greenplum Database, the old savepoint is kept, though only the more recent one will be used when rolling back or releasing. (Releasing the newer savepoint will cause the older one to again become accessible to *ROLLBACK TO SAVEPOINT* and *RELEASE SAVEPOINT*.) Otherwise, SAVEPOINT is fully SQL conforming.

See Also

BEGIN, *COMMIT*, *ROLLBACK*, *RELEASE SAVEPOINT*, *ROLLBACK TO SAVEPOINT*

SELECT

Retrieves rows from a table or view.

Synopsis

```
[ WITH [ RECURSIVE[1] ] with_query [, ...] ]
SELECT [ALL | DISTINCT [ON (expression [, ...])]]
  * | expression [[AS] output_name] [, ...]
  [FROM from_item [, ...]]
  [WHERE condition]
  [GROUP BY grouping_element [, ...]]
  [HAVING condition [, ...]]
  [WINDOW window_name AS (window_definition) [, ...] ]
  [{UNION | INTERSECT | EXCEPT} [ALL | DISTINCT] select]
  [ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}]
 [, ...]]
  [LIMIT {count | ALL}]
  [OFFSET start [ ROW | ROWS ] ]
  [FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY]
  [FOR {UPDATE | NO KEY UPDATE | SHARE | KEY SHARE} [OF table_name [, ...]]
 [NOWAIT] [...]]

TABLE { [ ONLY ] table_name [ * ] | with_query_name }
```

where *with_query:* is:

```
  with_query_name [( column_name [, ...] )] AS ( select | values | insert
 | update | delete )
```

where *from_item* can be one of:

```
[ONLY] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
( select ) [ AS ] alias [( column_alias [, ...] ) ]
with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
```

```
function_name ( [ argument [, ...] ] )
            [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias
 [, ...] ) ] ]
function_name ( [ argument [, ...] ] ) [ AS ] alias ( column_definition
 [, ...] )
function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
ROWS FROM( function_name ( [ argument [, ...] ] ) [ AS ( column_definition
 [, ...] ) ] [, ...] )
            [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias
 [, ...] ) ] ]
from_item [ NATURAL ] join_type from_item
          [ ON join_condition | USING ( join_column [, ...] ) ]
```

where *grouping_element* can be one of:

```
  ()
  expression
  ROLLUP (expression [,...])
  CUBE (expression [,...])
  GROUPING SETS ((grouping_element [, ...]))
```

where *window_definition* is:

```
  [existing_window_name]
  [PARTITION BY expression [, ...]]
  [ORDER BY expression [ASC | DESC | USING operator]
    [NULLS {FIRST | LAST}] [, ...]]
  [{ RANGE | ROWS} frame_start
     | {RANGE | ROWS} BETWEEN frame_start AND frame_end
```

where *frame_start* and *frame_end* can be one of:

```
  UNBOUNDED PRECEDING
  value PRECEDING
  CURRENT ROW
  value FOLLOWING
  UNBOUNDED FOLLOWING
```

Note: [1]The `RECURSIVE` keyword is A Beta feature.

Description

`SELECT` retrieves rows from zero or more tables. The general processing of `SELECT` is as follows:

1. All queries in the `WITH` clause are computed. These effectively serve as temporary tables that can be referenced in the `FROM` list.
2. All elements in the `FROM` list are computed. (Each element in the `FROM` list is a real or virtual table.) If more than one element is specified in the `FROM` list, they are cross-joined together.
3. If the `WHERE` clause is specified, all rows that do not satisfy the condition are eliminated from the output.
4. If the `GROUP BY` clause is specified, or if there are aggregate function calls, the output is combined into groups of rows that match on one or more values, and the results of aggregate functions are computed. If the `HAVING` clause is present, it eliminates groups that do not satisfy the given condition.
5. The actual output rows are computed using the `SELECT` output expressions for each selected row or row group.
6. `SELECT DISTINCT` eliminates duplicate rows from the result. `SELECT DISTINCT ON` eliminates rows that match on all the specified expressions. `SELECT ALL` (the default) will return all candidate rows, including duplicates.
7. If a window expression is specified (and optional `WINDOW` clause), the output is organized according to the positional (row) or value-based (range) window frame.

8. The actual output rows are computed using the SELECT output expressions for each selected row.
9. Using the operators UNION, INTERSECT, and EXCEPT, the output of more than one SELECT statement can be combined to form a single result set. The UNION operator returns all rows that are in one or both of the result sets. The INTERSECT operator returns all rows that are strictly in both result sets. The EXCEPT operator returns the rows that are in the first result set but not in the second. In all three cases, duplicate rows are eliminated unless ALL is specified. The noise word DISTINCT can be added to explicitly specify eliminating duplicate rows. Notice that DISTINCT is the default behavior here, even though ALL is the default for SELECT itself.
10. If the ORDER BY clause is specified, the returned rows are sorted in the specified order. If ORDER BY is not given, the rows are returned in whatever order the system finds fastest to produce.
11. If the LIMIT (or FETCH FIRST) or OFFSET clause is specified, the SELECT statement only returns a subset of the result rows.
12. If FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE, or FOR KEY SHARE is specified, the SELECT statement locks the entire table against concurrent updates.

You must have SELECT privilege on each column used in a SELECT command. The use of FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, or FOR KEY SHARE requires UPDATE privilege as well (for at least one column of each table so selected).

Parameters

The WITH Clause

The optional WITH clause allows you to specify one or more subqueries that can be referenced by name in the primary query. The subqueries effectively act as temporary tables or views for the duration of the primary query. Each subquery can be a SELECT, INSERT, UPDATE, or DELETE statement. When writing a data-modifying statement (INSERT, UPDATE, or DELETE) in WITH, it is usual to include a RETURNING clause. It is the output of RETURNING, *not* the underlying table that the statement modifies, that forms the temporary table that is read by the primary query. If RETURNING is omitted, the statement is still executed, but it produces no output so it cannot be referenced as a table by the primary query.

For a SELECT command that includes a WITH clause, the clause can contain at most a single clause that modifies table data (INSERT, UPDATE or DELETE command).

A *with_query_name* without schema qualification must be specified for each query in the WITH clause. Optionally, a list of column names can be specified; if the list of column names is omitted, the names are inferred from the subquery. The primary query and the WITH queries are all (notionally) executed at the same time.

If RECURSIVE is specified, it allows a SELECT subquery to reference itself by name. Such a subquery has the general form

```
non_recursive_term UNION [ALL | DISTINCT] recursive_term
```

where the recursive self-reference appears on the right-hand side of the UNION. Only one recursive self-reference is permitted per query. Recursive data-modifying statements are not supported, but you can use the results of a recursive SELECT query in a data-modifying statement.

If the RECURSIVE keyword is specified, the WITH queries need not be ordered: a query can reference another query that is later in the list. However, circular references, or mutual recursion, are not supported.

Without the RECURSIVE keyword, WITH queries can only reference sibling WITH queries that are earlier in the WITH list.

WITH RECURSIVE limitations. These items are not supported:

- A recursive WITH clause that contains the following in the *recursive_term*.
  - Subqueries with a self-reference
  - DISTINCT clause
  - GROUP BY clause

    - A window function
- A recursive `WITH` clause where the *with_query_name* is a part of a set operation.

Following is an example of the set operation limitation. This query returns an error because the set operation `UNION` contains a reference to the table `foo`.

```
WITH RECURSIVE foo(i) AS (
    SELECT 1
  UNION ALL
    SELECT i+1 FROM (SELECT * FROM foo UNION SELECT 0) bar
)
SELECT * FROM foo LIMIT 5;
```

This recursive CTE is allowed because the set operation `UNION` does not have a reference to the CTE `foo`.

```
WITH RECURSIVE foo(i) AS (
    SELECT 1
  UNION ALL
    SELECT i+1 FROM (SELECT * FROM bar UNION SELECT 0) bar, foo
    WHERE foo.i = bar.a
)
SELECT * FROM foo LIMIT 5;
```

A key property of `WITH` queries is that they are evaluated only once per execution of the primary query, even if the primary query refers to them more than once. In particular, data-modifying statements are guaranteed to be executed once and only once, regardless of whether the primary query reads all or any of their output.

The primary query and the `WITH` queries are all (notionally) executed at the same time. This implies that the effects of a data-modifying statement in `WITH` cannot be seen from other parts of the query, other than by reading its `RETURNING` output. If two such data-modifying statements attempt to modify the same row, the results are unspecified.

See *WITH Queries (Common Table Expressions)* in the *Greenplum Database Administrator Guide* for additional information.

The SELECT List

The `SELECT` list (between the key words `SELECT` and `FROM`) specifies expressions that form the output rows of the `SELECT` statement. The expressions can (and usually do) refer to columns computed in the `FROM` clause.

An *expression* in the `SELECT` list can be a constant value, a column reference, an operator invocation, a function call, an aggregate expression, a window expression, a scalar subquery, and so on. A number of constructs can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator. For information about SQL value expressions and function calls, see "Querying Data" in the *Greenplum Database Administrator Guide*.

Just as in a table, every output column of a `SELECT` has a name. In a simple `SELECT` this name is just used to label the column for display, but when the `SELECT` is a sub-query of a larger query, the name is seen by the larger query as the column name of the virtual table produced by the sub-query. To specify the name to use for an output column, write `AS` *output_name* after the column's expression. (You can omit `AS`, but only if the desired output name does not match any SQL keyword. For protection against possible future keyword additions, you can always either write `AS` or double-quote the output name.) If you do not specify a column name, Greenplum Database chooses a name is automatically. If the column's expression is a simple column reference then the chosen name is the same as that column's name. In more complex cases, a function or type name may be used, or the system may fall back on a generated name such as `?column?` or `column`*N*.

An output column's name can be used to refer to the column's value in ORDER BY and GROUP BY clauses, but not in the WHERE or HAVING clauses; there you must write out the expression instead.

Instead of an expression, * can be written in the output list as a shorthand for all the columns of the selected rows. Also, you can write *table_name*.* as a shorthand for the columns coming from just that table. In these cases it is not possible to specify new names with AS; the output column names will be the same as the table columns' names.

The DISTINCT Clause

If SELECT DISTINCT is specified, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). SELECT ALL specifies the opposite: all rows are kept; that is the default.

SELECT DISTINCT ON (*expression* [, ...]) keeps only the first row of each set of rows where the given expressions evaluate to equal. The DISTINCT ON expressions are interpreted using the same rules as for ORDER BY (see above). Note that the "first row" of each set is unpredictable unless ORDER BY is used to ensure that the desired row appears first. For example:

```
SELECT DISTINCT ON (location) location, time, report
    FROM weather_reports
    ORDER BY location, time DESC;
```

retrieves the most recent weather report for each location. But if we had not used ORDER BY to force descending order of time values for each location, we'd have gotten a report from an unpredictable time for each location.

The DISTINCT ON expression(s) must match the leftmost ORDER BY expression(s). The ORDER BY clause will normally contain additional expression(s) that determine the desired precedence of rows within each DISTINCT ON group.

The FROM Clause

The FROM clause specifies one or more source tables for the SELECT. If multiple sources are specified, the result is the Cartesian product (cross join) of all the sources. But usually qualification conditions are added (via WHERE) to restrict the returned rows to a small subset of the Cartesian product. The FROM clause can contain the following elements:

table_name

The name (optionally schema-qualified) of an existing table or view. If ONLY is specified, only that table is scanned. If ONLY is not specified, the table and all its descendant tables (if any) are scanned.

alias

A substitute name for the FROM item containing the alias. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). When an alias is provided, it completely hides the actual name of the table or function; for example given FROM foo AS f, the remainder of the SELECT must refer to this FROM item as f not foo. If an alias is written, a column alias list can also be written to provide substitute names for one or more columns of the table.

select

A sub-SELECT can appear in the FROM clause. This acts as though its output were created as a temporary table for the duration of this single SELECT command. Note that the sub-SELECT must be surrounded by parentheses, and an alias must be provided for it. A *VALUES* command can also be used here. See "Non-standard Clauses" in the *Compatibility* section for limitations of using correlated sub-selects in Greenplum Database.

with_query_name

A *with_query* is referenced in the FROM clause by specifying its *with_query_name*, just as though the name were a table name. The *with_query_name* cannot contain a schema qualifier. An alias can be provided in the same way as for a table.

The *with_query* hides a table of the same name for the purposes of the primary query. If necessary, you can refer to a table of the same name by qualifying the table name with the schema.

function_name

Function calls can appear in the FROM clause. (This is especially useful for functions that return result sets, but any function can be used.) This acts as though its output were created as a temporary table for the duration of this single SELECT command. An alias may also be used. If an alias is written, a column alias list can also be written to provide substitute names for one or more attributes of the function's composite return type. If the function has been defined as returning the record data type, then an alias or the key word AS must be present, followed by a column definition list in the form `( column_name data_type [, ... ] )`. The column definition list must match the actual number and types of columns returned by the function.

join_type

One of:

- **[INNER] JOIN**
- **LEFT [OUTER] JOIN**
- **RIGHT [OUTER] JOIN**
- **FULL [OUTER] JOIN**
- **CROSS JOIN**

For the INNER and OUTER join types, a join condition must be specified, namely exactly one of NATURAL, `ON join_condition`, or `USING ( join_column [, ...])`. See below for the meaning. For CROSS JOIN, none of these clauses may appear.

A JOIN clause combines two FROM items, which for convenience we will refer to as "tables", though in reality they can be any type of FROM item. Use parentheses if necessary to determine the order of nesting. In the absence of parentheses, JOINs nest left-to-right. In any case JOIN binds more tightly than the commas separating FROM-list items.

CROSS JOIN and INNER JOIN produce a simple Cartesian product, the same result as you get from listing the two tables at the top level of FROM, but restricted by the join condition (if any). CROSS JOIN is equivalent to INNER JOIN ON(TRUE), that is, no rows are removed by qualification. These join types are just a notational convenience, since they do nothing you could not do with plain FROM and WHERE.

LEFT OUTER JOIN returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its join condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting null values for the right-hand columns. Note that only the JOIN clause's own condition is considered while deciding which rows have matches. Outer conditions are applied afterwards.

Conversely, RIGHT OUTER JOIN returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left). This is just a notational convenience, since you could convert it to a LEFT OUTER JOIN by switching the left and right tables.

FULL OUTER JOIN returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

ON *join_condition*

join_condition is an expression resulting in a value of type `boolean` (similar to a `WHERE` clause) that specifies which rows in a join are considered to match.

`USING (join_column [, ...])`

A clause of the form `USING ( a, b, ... )` is shorthand for `ON left_table.a = right_table.a AND left_table.b = right_table.b ...`. Also, `USING` implies that only one of each pair of equivalent columns will be included in the join output, not both.

`NATURAL`

`NATURAL` is shorthand for a `USING` list that mentions all columns in the two tables that have the same names. If there are no common column names, `NATURAL` is equivalent to `ON TRUE`.

The WHERE Clause

The optional `WHERE` clause has the general form:

```
WHERE condition
```

where *condition* is any expression that evaluates to a result of type `boolean`. Any row that does not satisfy this condition will be eliminated from the output. A row satisfies the condition if it returns true when the actual row values are substituted for any variable references.

The GROUP BY Clause

The optional `GROUP BY` clause has the general form:

```
GROUP BY grouping_element [, ...]
```

where *grouping_element* can be one of:

```
()
expression
ROLLUP (expression [,...])
CUBE (expression [,...])
GROUPING SETS ((grouping_element [, ...]))
```

`GROUP BY` will condense into a single row all selected rows that share the same values for the grouped expressions. *expression* can be an input column name, or the name or ordinal number of an output column (`SELECT` list item), or an arbitrary expression formed from input-column values. In case of ambiguity, a `GROUP BY` name will be interpreted as an input-column name rather than an output column name.

Aggregate functions, if any are used, are computed across all rows making up each group, producing a separate value for each group. (If there are aggregate functions but no `GROUP BY` clause, the query is treated as having a single group comprising all the selected rows.) The set of rows fed to each aggregate function can be further filtered by attaching a `FILTER` clause to the aggregate function call. When a `FILTER` clause is present, only those rows matching it are included in the input to that aggregate function. See *Aggregate Expressions*.

When `GROUP BY` is present, or any aggregate functions are present, it is not valid for the `SELECT` list expressions to refer to ungrouped columns except within aggregate functions or when the ungrouped column is functionally dependent on the grouped columns, since there would otherwise be more than one possible value to return for an ungrouped column. A functional dependency exists if the grouped columns (or a subset thereof) are the primary key of the table containing the ungrouped column.

Keep in mind that all aggregate functions are evaluated before evaluating any "scalar" expressions in the `HAVING` clause or `SELECT` list. This means that, for example, a `CASE` expression cannot be used to skip evaluation of an aggregate function; see *Expression Evaluation Rules*.

Greenplum Database has the following additional OLAP grouping extensions (often referred to as *supergroups*):

ROLLUP

A ROLLUP grouping is an extension to the GROUP BY clause that creates aggregate subtotals that roll up from the most detailed level to a grand total, following a list of grouping columns (or expressions). ROLLUP takes an ordered list of grouping columns, calculates the standard aggregate values specified in the GROUP BY clause, then creates progressively higher-level subtotals, moving from right to left through the list. Finally, it creates a grand total. A ROLLUP grouping can be thought of as a series of grouping sets. For example:

```
GROUP BY ROLLUP (a,b,c)
```

is equivalent to:

```
GROUP BY GROUPING SETS( (a,b,c), (a,b), (a), () )
```

Notice that the *n* elements of a ROLLUP translate to *n*+1 grouping sets. Also, the order in which the grouping expressions are specified is significant in a ROLLUP.

CUBE

A CUBE grouping is an extension to the GROUP BY clause that creates subtotals for all of the possible combinations of the given list of grouping columns (or expressions). In terms of multidimensional analysis, CUBE generates all the subtotals that could be calculated for a data cube with the specified dimensions. For example:

```
GROUP BY CUBE (a,b,c)
```

is equivalent to:

```
GROUP BY GROUPING SETS( (a,b,c), (a,b), (a,c), (b,c), (a),
(b), (c), () )
```

Notice that *n* elements of a CUBE translate to 2n grouping sets. Consider using CUBE in any situation requiring cross-tabular reports. CUBE is typically most suitable in queries that use columns from multiple dimensions rather than columns representing different levels of a single dimension. For instance, a commonly requested cross-tabulation might need subtotals for all the combinations of month, state, and product.

GROUPING SETS

You can selectively specify the set of groups that you want to create using a GROUPING SETS expression within a GROUP BY clause. This allows precise specification across multiple dimensions without computing a whole ROLLUP or CUBE. For example:

```
GROUP BY GROUPING SETS( (a,c), (a,b) )
```

If using the grouping extension clauses ROLLUP, CUBE, or GROUPING SETS, two challenges arise. First, how do you determine which result rows are subtotals, and then the exact level of aggregation for a given subtotal. Or, how do you differentiate between result rows that contain both stored NULL values and "NULL" values created by the ROLLUP or CUBE. Secondly, when duplicate grouping sets are specified in the GROUP BY clause, how do you determine which result rows are duplicates? There are two additional grouping functions you can use in the SELECT list to help with this:

- **grouping(column [, ...])** — The grouping function can be applied to one or more grouping attributes to distinguish super-aggregated rows from regular grouped rows. This can be helpful in distinguishing a "NULL" representing the set of all values in a super-aggregated row from a NULL value in a regular row. Each argument in this function produces a bit — either 1 or 0, where 1 means the result row is super-aggregated, and 0 means the result row is from a regular grouping. The grouping

function returns an integer by treating these bits as a binary number and then converting it to a base-10 integer.

- **group_id()** — For grouping extension queries that contain duplicate grouping sets, the `group_id` function is used to identify duplicate rows in the output. All *unique* grouping set output rows will have a group_id value of 0. For each duplicate grouping set detected, the `group_id` function assigns a group_id number greater than 0. All output rows in a particular duplicate grouping set are identified by the same group_id number.

The WINDOW Clause

The optional `WINDOW` clause specifies the behavior of window functions appearing in the query's `SELECT` list or `ORDER BY` clause. These functions can reference the `WINDOW` clause entries by name in their `OVER` clauses. A `WINDOW` clause entry does not have to be referenced anywhere, however; if it is not used in the query it is simply ignored. It is possible to use window functions without any `WINDOW` clause at all, since a window function call can specify its window definition directly in its `OVER` clause. However, the `WINDOW` clause saves typing when the same window definition is needed for more than one window function.

For example:

```
SELECT vendor, rank() OVER (mywindow) FROM sale
GROUP BY vendor
WINDOW mywindow AS (ORDER BY sum(prc*qty));
```

A `WINDOW` clause has this general form:

```
WINDOW window_name AS (window_definition)
```

where *window_name* is a name that can be referenced from `OVER` clauses or subsequent window definitions, and *window_definition* is:

```
[existing_window_name]
[PARTITION BY expression [, ...]]
[ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}]
 [, ...] ]
[frame_clause]
```

existing_window_name

If an *existing_window_name* is specified it must refer to an earlier entry in the `WINDOW` list; the new window copies its partitioning clause from that entry, as well as its ordering clause if any. The new window cannot specify its own `PARTITION BY` clause, and it can specify `ORDER BY` only if the copied window does not have one. The new window always uses its own frame clause; the copied window must not specify a frame clause.

PARTITION BY

The `PARTITION BY` clause organizes the result set into logical groups based on the unique values of the specified expression. The elements of the `PARTITION BY` clause are interpreted in much the same fashion as elements of a `GROUP BY` clause, except that they are always simple expressions and never the name or number of an output column. Another difference is that these expressions can contain aggregate function calls, which are not allowed in a regular `GROUP BY` clause. They are allowed here because windowing occurs after grouping and aggregation. When used with window functions, the functions are applied to each partition independently. For example, if you follow `PARTITION BY` with a column name, the result set is partitioned by the distinct values of that column. If omitted, the entire result set is considered one partition.

Similarly, the elements of the `ORDER BY` list are interpreted in much the same fashion as elements of an `ORDER BY` clause, except that the expressions are always taken as simple expressions and never the name or number of an output column.

ORDER BY

The elements of the ORDER BY clause define how to sort the rows in each partition of the result set. If omitted, rows are returned in whatever order is most efficient and may vary. **Note:** Columns of data types that lack a coherent ordering, such as time, are not good candidates for use in the ORDER BY clause of a window specification. Time, with or without time zone, lacks a coherent ordering because addition and subtraction do not have the expected effects. For example, the following is not generally true: x::time < x::time + '2 hour'::interval

frame_clause

The optional *frame_clause* defines the *window frame* for window functions that depend on the frame (not all do). The window frame is a set of related rows for each row of the query (called the *current row*). The *frame_clause* can be one of

```
{ RANGE | ROWS } frame_start
{ RANGE | ROWS } BETWEEN frame_start AND frame_end
```

where *frame_start* and *frame_end* can be one of

- UNBOUNDED PRECEDING
- *value* PRECEDING
- CURRENT ROW
- *value* FOLLOWING
- UNBOUNDED FOLLOWING

If *frame_end* is omitted it defaults to CURRENT ROW. Restrictions are that *frame_start* cannot be UNBOUNDED FOLLOWING, *frame_end* cannot be UNBOUNDED PRECEDING, and the *frame_end* choice cannot appear earlier in the above list than the *frame_start* choice — for example RANGE BETWEEN CURRENT ROW AND *value* PRECEDING is not allowed.

The default framing option is RANGE UNBOUNDED PRECEDING, which is the same as RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW; it sets the frame to be all rows from the partition start up through the current row's last peer (a row that ORDER BY considers equivalent to the current row, or all rows if there is no ORDER BY). In general, UNBOUNDED PRECEDING means that the frame starts with the first row of the partition, and similarly UNBOUNDED FOLLOWING means that the frame ends with the last row of the partition (regardless of RANGE or ROWS mode). In ROWS mode, CURRENT ROW means that the frame starts or ends with the current row; but in RANGE mode it means that the frame starts or ends with the current row's first or last peer in the ORDER BY ordering. The *value* PRECEDING and *value* FOLLOWING cases are currently only allowed in ROWS mode. They indicate that the frame starts or ends with the row that many rows before or after the current row. *value* must be an integer expression not containing any variables, aggregate functions, or window functions. The value must not be null or negative; but it can be zero, which selects the current row itself.

Beware that the ROWS options can produce unpredictable results if the ORDER BY ordering does not order the rows uniquely. The RANGE options are designed to ensure that rows that are peers in the ORDER BY ordering are treated alike; all peer rows will be in the same frame.

Use either a ROWS or RANGE clause to express the bounds of the window. The window bound can be one, many, or all rows of a partition. You can express the bound of the window either in terms of a range of data values offset from the value in the current row (RANGE), or in terms of the number of rows offset from the current row (ROWS). When using the RANGE clause, you must also use an ORDER BY clause. This is because the calculation performed to produce the window requires that the values be sorted. Additionally, the ORDER BY clause cannot contain more than one expression, and the expression must

result in either a date or a numeric value. When using the ROWS or RANGE clauses, if you specify only a starting row, the current row is used as the last row in the window.

PRECEDING — The PRECEDING clause defines the first row of the window using the current row as a reference point. The starting row is expressed in terms of the number of rows preceding the current row. For example, in the case of ROWS framing, 5 PRECEDING sets the window to start with the fifth row preceding the current row. In the case of RANGE framing, it sets the window to start with the first row whose ordering column value precedes that of the current row by 5 in the given order. If the specified order is ascending by date, this will be the first row within 5 days before the current row. UNBOUNDED PRECEDING sets the first row in the window to be the first row in the partition.

BETWEEN — The BETWEEN clause defines the first and last row of the window, using the current row as a reference point. First and last rows are expressed in terms of the number of rows preceding and following the current row, respectively. For example, BETWEEN 3 PRECEDING AND 5 FOLLOWING sets the window to start with the third row preceding the current row, and end with the fifth row following the current row. Use BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING to set the first and last rows in the window to be the first and last row in the partition, respectively. This is equivalent to the default behavior if no ROW or RANGE clause is specified.

FOLLOWING — The FOLLOWING clause defines the last row of the window using the current row as a reference point. The last row is expressed in terms of the number of rows following the current row. For example, in the case of ROWS framing, 5 FOLLOWING sets the window to end with the fifth row following the current row. In the case of RANGE framing, it sets the window to end with the last row whose ordering column value follows that of the current row by 5 in the given order. If the specified order is ascending by date, this will be the last row within 5 days after the current row. Use UNBOUNDED FOLLOWING to set the last row in the window to be the last row in the partition.

If you do not specify a ROW or a RANGE clause, the window bound starts with the first row in the partition (UNBOUNDED PRECEDING) and ends with the current row (CURRENT ROW) if ORDER BY is used. If an ORDER BY is not specified, the window starts with the first row in the partition (UNBOUNDED PRECEDING) and ends with last row in the partition (UNBOUNDED FOLLOWING).

The HAVING Clause

The optional HAVING clause has the general form:

```
HAVING condition
```

where *condition* is the same as specified for the WHERE clause. HAVING eliminates group rows that do not satisfy the condition. HAVING is different from WHERE: WHERE filters individual rows before the application of GROUP BY, while HAVING filters group rows created by GROUP BY. Each column referenced in *condition* must unambiguously reference a grouping column, unless the reference appears within an aggregate function or the ungrouped column is functionally dependent on the grouping columns.

The presence of HAVING turns a query into a grouped query even if there is no GROUP BY clause. This is the same as what happens when the query contains aggregate functions but no GROUP BY clause. All the selected rows are considered to form a single group, and the SELECT list and HAVING clause can only reference table columns from within aggregate functions. Such a query will emit a single row if the HAVING condition is true, zero rows if it is not true.

The UNION Clause

The UNION clause has this general form:

```
select_statement UNION [ALL | DISTINCT] select_statement
```

where *select_statement* is any SELECT statement without an ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, or FOR KEY SHARE clause. (ORDER BY and LIMIT can be attached to a subquery expression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to apply to the result of the UNION, not to its right-hand input expression.)

The UNION operator computes the set union of the rows returned by the involved SELECT statements. A row is in the set union of two result sets if it appears in at least one of the result sets. The two SELECT statements that represent the direct operands of the UNION must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of UNION does not contain any duplicate rows unless the ALL option is specified. ALL prevents elimination of duplicates. (Therefore, UNION ALL is usually significantly quicker than UNION; use ALL when you can.) DISTINCT can be written to explicitly specify the default behavior of eliminating duplicate rows.

Multiple UNION operators in the same SELECT statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, and FOR KEY SHARE cannot be specified either for a UNION result or for any input of a UNION.

The INTERSECT Clause

The INTERSECT clause has this general form:

```
select_statement INTERSECT [ALL | DISTINCT] select_statement
```

where *select_statement* is any SELECT statement without an ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, or FOR KEY SHARE clause.

The INTERSECT operator computes the set intersection of the rows returned by the involved SELECT statements. A row is in the intersection of two result sets if it appears in both result sets.

The result of INTERSECT does not contain any duplicate rows unless the ALL option is specified. With ALL, a row that has *m* duplicates in the left table and *n* duplicates in the right table will appear min(*m*, *n*) times in the result set. DISTINCT can be written to explicitly specify the default behavior of eliminating duplicate rows.

Multiple INTERSECT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. INTERSECT binds more tightly than UNION. That is, A UNION B INTERSECT C will be read as A UNION (B INTERSECT C).

Currently, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, and FOR KEY SHARE cannot be specified either for an INTERSECT result or for any input of an INTERSECT.

The EXCEPT Clause

The EXCEPT clause has this general form:

```
select_statement EXCEPT [ALL | DISTINCT] select_statement
```

where *select_statement* is any SELECT statement without an ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, or FOR KEY SHARE clause.

The EXCEPT operator computes the set of rows that are in the result of the left SELECT statement but not in the result of the right one.

The result of EXCEPT does not contain any duplicate rows unless the ALL option is specified. With ALL, a row that has *m* duplicates in the left table and *n* duplicates in the right table will appear max(*m*-*n*,0) times in the result set. DISTINCT can be written to explicitly specify the default behavior of eliminating duplicate rows.

Multiple EXCEPT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. EXCEPT binds at the same level as UNION.

Currently, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, and FOR KEY SHARE cannot be specified either for an EXCEPT result or for any input of an EXCEPT.

The ORDER BY Clause

The optional ORDER BY clause has this general form:

```
ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}]
 [,...]
```

where *expression* can be the name or ordinal number of an output column (SELECT list item), or it can be an arbitrary expression formed from input-column values.

The ORDER BY clause causes the result rows to be sorted according to the specified expressions. If two rows are equal according to the left-most expression, they are compared according to the next expression and so on. If they are equal according to all specified expressions, they are returned in an implementation-dependent order.

The ordinal number refers to the ordinal (left-to-right) position of the output column. This feature makes it possible to define an ordering on the basis of a column that does not have a unique name. This is never absolutely necessary because it is always possible to assign a name to an output column using the AS clause.

It is also possible to use arbitrary expressions in the ORDER BY clause, including columns that do not appear in the SELECT output list. Thus the following statement is valid:

```
SELECT name FROM distributors ORDER BY code;
```

A limitation of this feature is that an ORDER BY clause applying to the result of a UNION, INTERSECT, or EXCEPT clause may only specify an output column name or number, not an expression.

If an ORDER BY expression is a simple name that matches both an output column name and an input column name, ORDER BY will interpret it as the output column name. This is the opposite of the choice that GROUP BY will make in the same situation. This inconsistency is made to be compatible with the SQL standard.

Optionally one may add the key word ASC (ascending) or DESC (descending) after any expression in the ORDER BY clause. If not specified, ASC is assumed by default. Alternatively, a specific ordering operator name may be specified in the USING clause. ASC is usually equivalent to USING < and DESC is usually equivalent to USING >. (But the creator of a user-defined data type can define exactly what the default sort ordering is, and it might correspond to operators with other names.)

If NULLS LAST is specified, null values sort after all non-null values; if NULLS FIRST is specified, null values sort before all non-null values. If neither is specified, the default behavior is NULLS LAST when ASC is specified or implied, and NULLS FIRST when DESC is specified (thus, the default is to act as though nulls are larger than non-nulls). When USING is specified, the default nulls ordering depends upon whether the operator is a less-than or greater-than operator.

Note that ordering options apply only to the expression they follow; for example ORDER BY x, y DESC does not mean the same thing as ORDER BY x DESC, y DESC.

Character-string data is sorted according to the locale-specific collation order that was established when the database was created.

Character-string data is sorted according to the collation that applies to the column being sorted. That can be overridden as needed by including a COLLATE clause in the *expression*, for example ORDER BY mycolumn COLLATE "en_US". For information about defining collations, see *CREATE COLLATION*.

The LIMIT Clause

The LIMIT clause consists of two independent sub-clauses:

```
LIMIT {count | ALL}
```

```
OFFSET start
```

where *count* specifies the maximum number of rows to return, while *start* specifies the number of rows to skip before starting to return rows. When both are specified, start rows are skipped before starting to count the `count` rows to be returned.

If the `count` expression evaluates to NULL, it is treated as `LIMIT ALL`, that is, no limit. If `start` evaluates to NULL, it is treated the same as `OFFSET 0`.

SQL:2008 introduced a different syntax to achieve the same result, which Greenplum Database also supports. It is:

```
OFFSET start [ ROW | ROWS ]
            FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY
```

In this syntax, the *start* or *count* value is required by the standard to be a literal constant, a parameter, or a variable name; as a Greenplum Database extension, other expressions are allowed, but will generally need to be enclosed in parentheses to avoid ambiguity. If *count* is omitted in a `FETCH` clause, it defaults to 1. `ROW` and `ROWS` as well as `FIRST` and `NEXT` are noise words that don't influence the effects of these clauses. According to the standard, the `OFFSET` clause must come before the `FETCH` clause if both are present; but Greenplum Database allows either order.

When using `LIMIT`, it is a good idea to use an `ORDER BY` clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows — you may be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? You don't know what ordering unless you specify `ORDER BY`.

The query optimizer takes `LIMIT` into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you use for `LIMIT` and `OFFSET`. Thus, using different `LIMIT/OFFSET` values to select different subsets of a query result will give inconsistent results unless you enforce a predictable result ordering with `ORDER BY`. This is not a defect; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless `ORDER BY` is used to constrain the order.

The Locking Clause

`FOR UPDATE`, `FOR NO KEY UPDATE`, `FOR SHARE` and `FOR KEY SHARE` are *locking clauses*; they affect how `SELECT` locks rows as they are obtained from the table.

The locking clause has the general form

```
FOR lock_strength [OF table_name [ , ... ] ] [ NOWAIT ]
```

where *lock_strength* can be one of

- `FOR UPDATE` - Locks the table with an `EXCLUSIVE` lock.
- `FOR NO KEY UPDATE` - Locks the table with an `EXCLUSIVE` lock.
- `FOR SHARE` - Locks the table with a `ROW SHARE` lock.
- `FOR KEY SHARE` - Locks the table with a `ROW SHARE` lock.

> **Note:** By default Greenplum Database acquires the more restrictive `EXCLUSIVE` lock (rather than `ROW EXCLUSIVE` in PostgreSQL) for `UPDATE`, `DELETE`, and `SELECT...FOR UPDATE` operations on heap tables. When the Global Deadlock Detector is enabled the lock mode for `UPDATE` and `DELETE` operations on heap tables is `ROW EXCLUSIVE`. See *Global Deadlock Detector*. Greenplum always holds a table-level lock with `SELECT...FOR UPDATE` statements.

For more information on each row-level lock mode, refer to *Explicit Locking* in the PostgreSQL documentation.

To prevent the operation from waiting for other transactions to commit, use the `NOWAIT` option. With `NOWAIT`, the statement reports an error, rather than waiting, if a selected row cannot be locked immediately. Note that `NOWAIT` only affects whether the `SELECT` statement waits to obtain row-level locks.

A required table-level lock is always taken in the ordinary way. For example, a `SELECT FOR UPDATE NOWAIT` statement will always wait for the required table-level lock; it behaves as if `NOWAIT` was omitted. You can use `LOCK` with the `NOWAIT` option first, if you need to acquire the table-level lock without waiting.

If specific tables are named in a locking clause, then only rows coming from those tables are locked; any other tables used in the `SELECT` are simply read as usual. A locking clause without a table list affects all tables used in the statement. If a locking clause is applied to a view or sub-query, it affects all tables used in the view or sub-query. However, these clauses do not apply to `WITH` queries referenced by the primary query. If you want row locking to occur within a `WITH` query, specify a locking clause within the `WITH` query.

Multiple locking clauses can be written if it is necessary to specify different locking behavior for different tables. If the same table is mentioned (or implicitly affected) by both more than one locking clause, then it is processed as if it was only specified by the strongest one. Similarly, a table is processed as `NOWAIT` if that is specified in any of the clauses affecting it.

The locking clauses cannot be used in contexts where returned rows cannot be clearly identified with individual table rows; for example they cannot be used with aggregation.

When a locking clause appears at the top level of a `SELECT` query, the rows that are locked are exactly those that are returned by the query; in the case of a join query, the rows locked are those that contribute to returned join rows. In addition, rows that satisfied the query conditions as of the query snapshot will be locked, although they will not be returned if they were updated after the snapshot and no longer satisfy the query conditions. If a `LIMIT` is used, locking stops once enough rows have been returned to satisfy the limit (but note that rows skipped over by `OFFSET` will get locked). Similarly, if a locking clause is used in a cursor's query, only rows actually fetched or stepped past by the cursor will be locked.

When locking clause appears in a sub-`SELECT`, the rows locked are those returned to the outer query by the sub-query. This might involve fewer rows than inspection of the sub-query alone would suggest, since conditions from the outer query might be used to optimize execution of the sub-query. For example,

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

will lock only rows having `col1 = 5`, even though that condition is not textually within the sub-query.

It is possible for a `SELECT` command running at the `READ COMMITTED` transaction isolation level and using `ORDER BY` and a locking clause to return rows out of order. This is because `ORDER BY` is applied first. The command sorts the result, but might then block trying to obtain a lock on one or more of the rows. Once the `SELECT` unblocks, some of the ordering column values might have been modified, leading to those rows appearing to be out of order (though they are in order in terms of the original column values). This can be worked around at need by placing the `FOR UPDATE/SHARE` clause in a sub-query, for example

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss ORDER BY column1;
```

Note that this will result in locking all rows of `mytable`, whereas `FOR UPDATE` at the top level would lock only the actually returned rows. This can make for a significant performance difference, particularly if the `ORDER BY` is combined with `LIMIT` or other restrictions. So this technique is recommended only if concurrent updates of the ordering columns are expected and a strictly sorted result is required.

At the `REPEATABLE READ` or `SERIALIZABLE` transaction isolation level this would cause a serialization failure (with a `SQLSTATE` of `40001`), so there is no possibility of receiving rows out of order under these isolation levels.

The TABLE Command

The command

```
TABLE name
```

is completely equivalent to

```
SELECT * FROM name
```

It can be used as a top-level command or as a space-saving syntax variant in parts of complex queries.

Examples

To join the table `films` with the table `distributors`:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind FROM
distributors d, films f WHERE f.did = d.did
```

To sum the column `length` of all films and group the results by `kind`:

```
SELECT kind, sum(length) AS total FROM films GROUP BY kind;
```

To sum the column `length` of all films, group the results by `kind` and show those group totals that are less than 5 hours:

```
SELECT kind, sum(length) AS total FROM films GROUP BY kind
HAVING sum(length) < interval '5 hours';
```

Calculate the subtotals and grand totals of all sales for movie `kind` and `distributor`.

```
SELECT kind, distributor, sum(prc*qty) FROM sales
GROUP BY ROLLUP(kind, distributor)
ORDER BY 1,2,3;
```

Calculate the rank of movie distributors based on total sales:

```
SELECT distributor, sum(prc*qty),
       rank() OVER (ORDER BY sum(prc*qty) DESC)
FROM sale
GROUP BY distributor ORDER BY 2 DESC;
```

The following two examples are identical ways of sorting the individual results according to the contents of the second column (`name`):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

The next example shows how to obtain the union of the tables `distributors` and `actors`, restricting the results to those that begin with the letter `W` in each table. Only distinct rows are wanted, so the key word `ALL` is omitted:

```
SELECT distributors.name FROM distributors WHERE
distributors.name LIKE 'W%' UNION SELECT actors.name FROM
actors WHERE actors.name LIKE 'W%';
```

This example shows how to use a function in the `FROM` clause, both with and without a column definition list:

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors
AS $$ SELECT * FROM distributors WHERE did = $1; $$ LANGUAGE
SQL;
SELECT * FROM distributors(111);

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS
```

```
$$ SELECT * FROM distributors WHERE did = $1; $$ LANGUAGE
SQL;
SELECT * FROM distributors_2(111) AS (dist_id int, dist_name
text);
```

This example uses a simple `WITH` clause:

```
WITH test AS (
  SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM test
UNION ALL
SELECT * FROM test;
```

This example uses the `WITH` clause to display per-product sales totals in only the top sales regions.

```
WITH regional_sales AS
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
  ), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales) FROM
       regional_sales)
  )
SELECT region, product, SUM(quantity) AS product_units,
   SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

The example could have been written without the `WITH` clause but would have required two levels of nested sub-`SELECT` statements.

This example uses the `WITH RECURSIVE` clause to find all subordinates (direct or indirect) of the employee Mary, and their level of indirectness, from a table that shows only direct subordinates:

```
WITH RECURSIVE employee_recursive(distance, employee_name, manager_name) AS
 (
    SELECT 1, employee_name, manager_name
    FROM employee
    WHERE manager_name = 'Mary'
  UNION ALL
    SELECT er.distance + 1, e.employee_name, e.manager_name
    FROM employee_recursive er, employee e
    WHERE er.employee_name = e.manager_name
  )
SELECT distance, employee_name FROM employee_recursive;
```

The typical form of recursive queries: an initial condition, followed by `UNION [ALL]`, followed by the recursive part of the query. Be sure that the recursive part of the query will eventually return no tuples, or else the query will loop indefinitely. See *WITH Queries (Common Table Expressions)* in the *Greenplum Database Administrator Guide* for more examples.

Compatibility

The `SELECT` statement is compatible with the SQL standard, but there are some extensions and some missing features.

Omitted FROM Clauses

Greenplum Database allows one to omit the FROM clause. It has a straightforward use to compute the results of simple expressions. For example:

```
SELECT 2+2;
```

Some other SQL databases cannot do this except by introducing a dummy one-row table from which to do the SELECT.

Note that if a FROM clause is not specified, the query cannot reference any database tables. For example, the following query is invalid:

```
SELECT distributors.* WHERE distributors.name = 'Westward';
```

In earlier releases, setting a server configuration parameter, *add_missing_from*, to true allowed Greenplum Database to add an implicit entry to the query's FROM clause for each table referenced by the query. This is no longer allowed.

Omitting the AS Key Word

In the SQL standard, the optional key word AS can be omitted before an output column name whenever the new column name is a valid column name (that is, not the same as any reserved keyword). Greenplum Database is slightly more restrictive: AS is required if the new column name matches any keyword at all, reserved or not. Recommended practice is to use AS or double-quote output column names, to prevent any possible conflict against future keyword additions.

In FROM items, both the standard and Greenplum Database allow AS to be omitted before an alias that is an unreserved keyword. But this is impractical for output column names, because of syntactic ambiguities.

ONLY and Inheritance

The SQL standard requires parentheses around the table name when writing ONLY, for example:

```
SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE ...
```

Greenplum Database considers these parentheses to be optional.

Greenplum Database allows a trailing * to be written to explicitly specify the non-ONLY behavior of including child tables. The standard does not allow this.

(These points apply equally to all SQL commands supporting the ONLY option.)

Namespace Available to GROUP BY and ORDER BY

In the SQL-92 standard, an ORDER BY clause may only use output column names or numbers, while a GROUP BY clause may only use expressions based on input column names. Greenplum Database extends each of these clauses to allow the other choice as well (but it uses the standard's interpretation if there is ambiguity). Greenplum Database also allows both clauses to specify arbitrary expressions. Note that names appearing in an expression are always taken as input-column names, not as output column names.

SQL:1999 and later use a slightly different definition which is not entirely upward compatible with SQL-92. In most cases, however, Greenplum Database interprets an ORDER BY or GROUP BY expression the same way SQL:1999 does.

Functional Dependencies

Greenplum Database recognizes functional dependency (allowing columns to be omitted from GROUP BY) only when a table's primary key is included in the GROUP BY list. The SQL standard specifies additional conditions that should be recognized.

LIMIT and OFFSET

The clauses LIMIT and OFFSET are Greenplum Database-specific syntax, also used by MySQL. The SQL:2008 standard has introduced the clauses OFFSET .. FETCH {FIRST|NEXT} ... for the same functionality, as shown above. This syntax is also used by IBM DB2. (Applications for Oracle frequently use

a workaround involving the automatically generated `rownum` column, which is not available in Greenplum Database, to implement the effects of these clauses.)

FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, and FOR KEY SHARE

Although `FOR UPDATE` appears in the SQL standard, the standard allows it only as an option of `DECLARE CURSOR`. Greenplum Database allows it in any `SELECT` query as well as in sub-`SELECT`s, but this is an extension. The `FOR NO KEY UPDATE`, `FOR SHARE`, and `FOR KEY SHARE` variants, as well as the `NOWAIT` option, do not appear in the standard.

Data-Modifying Statements in WITH

Greenplum Database allows `INSERT`, `UPDATE`, and `DELETE` to be used as `WITH` queries. This is not found in the SQL standard.

Nonstandard Clauses

The clause `DISTINCT ON` is not defined in the SQL standard.

Limited Use of STABLE and VOLATILE Functions

To prevent data from becoming out-of-sync across the segments in Greenplum Database, any function classified as `STABLE` or `VOLATILE` cannot be executed at the segment database level if it contains SQL or modifies the database in any way. See *CREATE FUNCTION* for more information.

See Also

EXPLAIN

SELECT INTO

Defines a new table from the results of a query.

Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ALL | DISTINCT [ON ( expression [, ...] )]]
    * | expression [AS output_name] [, ...]
    INTO [TEMPORARY | TEMP | UNLOGGED ] [TABLE] new_table
    [FROM from_item [, ...]]
    [WHERE condition]
    [GROUP BY expression [, ...]]
    [HAVING condition [, ...]]
    [{UNION | INTERSECT | EXCEPT} [ALL | DISTINCT ] select]
    [ORDER BY expression [ASC | DESC | USING operator] [NULLS {FIRST |
 LAST}] [, ...]]
    [LIMIT {count | ALL}]
    [OFFSET start [ ROW | ROWS ] ]
    [FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT]
    [...]]
```

Description

`SELECT INTO` creates a new table and fills it with data computed by a query. The data is not returned to the client, as it is with a normal `SELECT`. The new table's columns have the names and data types associated with the output columns of the `SELECT`.

Parameters

The majority of parameters for `SELECT INTO` are the same as *SELECT*.

TEMPORARY
TEMP

If specified, the table is created as a temporary table.

UNLOGGED

If specified, the table is created as an unlogged table. Data written to unlogged tables is not written to the write-ahead (WAL) log, which makes them considerably faster than ordinary tables. However, the contents of an unlogged table are not replicated to mirror segment instances. Also an unlogged table is not crash-safe. After a segment instance crash or unclean shutdown, the data for the unlogged table on that segment is truncated. Any indexes created on an unlogged table are automatically unlogged as well.

new_table

The name (optionally schema-qualified) of the table to be created.

Examples

Create a new table `films_recent` consisting of only recent entries from the table `films`:

```
SELECT * INTO films_recent FROM films WHERE date_prod >=
'2016-01-01';
```

Compatibility

The SQL standard uses `SELECT INTO` to represent selecting values into scalar variables of a host program, rather than creating a new table. The Greenplum Database usage of `SELECT INTO` to represent table creation is historical. It is best to use *CREATE TABLE AS* for this purpose in new applications.

See Also

SELECT, *CREATE TABLE AS*

SET

Changes the value of a Greenplum Database configuration parameter.

Synopsis

```
SET [SESSION | LOCAL] configuration_parameter {TO | =} value |
    'value' | DEFAULT}

SET [SESSION | LOCAL] TIME ZONE {timezone | LOCAL | DEFAULT}
```

Description

The `SET` command changes server configuration parameters. Any configuration parameter classified as a *session* parameter can be changed on-the-fly with `SET`. `SET` affects only the value used by the current session.

If `SET` or `SET SESSION` is issued within a transaction that is later aborted, the effects of the `SET` command disappear when the transaction is rolled back. Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another `SET`.

The effects of `SET LOCAL` last only till the end of the current transaction, whether committed or not. A special case is `SET` followed by `SET LOCAL` within a single transaction: the `SET LOCAL` value will be seen until the end of the transaction, but afterwards (if the transaction is committed) the `SET` value will take effect.

If `SET LOCAL` is used within a function that includes a `SET` option for the same configuration parameter (see *`CREATE FUNCTION`*), the effects of the `SET LOCAL` command disappear at function exit; the value in effect when the function was called is restored anyway. This allows `SET LOCAL` to be used for dynamic or repeated changes of a parameter within a function, while retaining the convenience of using the `SET` option to save and restore the caller's value. Note that a regular `SET` command overrides any surrounding function's `SET` option; its effects persist unless rolled back.

If you create a cursor with the `DECLARE` command in a transaction, you cannot use the `SET` command in the transaction until you close the cursor with the `CLOSE` command.

See *Server Configuration Parameters* for information about server parameters.

Parameters

`SESSION`

Specifies that the command takes effect for the current session. This is the default.

`LOCAL`

Specifies that the command takes effect for only the current transaction. After `COMMIT` or `ROLLBACK`, the session-level setting takes effect again. Note that `SET LOCAL` will appear to have no effect if it is executed outside of a transaction.

`configuration_parameter`

The name of a Greenplum Database configuration parameter. Only parameters classified as *session* can be changed with `SET`. See *Server Configuration Parameters* for details.

`value`

New value of parameter. Values can be specified as string constants, identifiers, numbers, or comma-separated lists of these. `DEFAULT` can be used to specify resetting the parameter to its default value. If specifying memory sizing or time units, enclose the value in single quotes.

`TIME ZONE`

`SET TIME ZONE` value is an alias for `SET timezone TO` *`value`*. The syntax `SET TIME ZONE` allows special syntax for the time zone specification. Here are examples of valid values:

`'PST8PDT'`

`'Europe/Rome'`

`-7` (time zone 7 hours west from UTC)

`INTERVAL '-08:00' HOUR TO MINUTE` (time zone 8 hours west from UTC).

`LOCAL`
`DEFAULT`

Set the time zone to your local time zone (that is, server's default value of *timezone*). See the *Time zone section of the PostgreSQL documentation* for more information about time zones in Greenplum Database.

Examples

Set the schema search path:

```
SET search_path TO my_schema, public;
```

Increase the segment host memory per query to 200 MB:

```
SET statement_mem TO '200MB';
```

Set the style of date to traditional POSTGRES with "day before month" input convention:

```
SET datestyle TO postgres, dmy;
```

Set the time zone for San Mateo, California (Pacific Time):

```
SET TIME ZONE 'PST8PDT';
```

Set the time zone for Italy:

```
SET TIME ZONE 'Europe/Rome';
```

Compatibility

`SET TIME ZONE` extends syntax defined in the SQL standard. The standard allows only numeric time zone offsets while Greenplum Database allows more flexible time-zone specifications. All other `SET` features are Greenplum Database extensions.

See Also

`RESET`, *`SHOW`*

SET CONSTRAINTS

Sets constraint check timing for the current transaction.

Note: Referential integrity syntax (foreign key constraints) is accepted but not enforced.

Synopsis

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

Description

`SET CONSTRAINTS` sets the behavior of constraint checking within the current transaction. `IMMEDIATE` constraints are checked at the end of each statement. `DEFERRED` constraints are not checked until transaction commit. Each constraint has its own `IMMEDIATE` or `DEFERRED` mode.

Upon creation, a constraint is given one of three characteristics: `DEFERRABLE INITIALLY DEFERRED`, `DEFERRABLE INITIALLY IMMEDIATE`, or `NOT DEFERRABLE`. The third class is always `IMMEDIATE` and is not affected by the `SET CONSTRAINTS` command. The first two classes start every transaction in the indicated mode, but their behavior can be changed within a transaction by `SET CONSTRAINTS`.

`SET CONSTRAINTS` with a list of constraint names changes the mode of just those constraints (which must all be deferrable). Each constraint name can be schema-qualified. The current schema search path is used to find the first matching name if no schema name is specified. `SET CONSTRAINTS ALL` changes the mode of all deferrable constraints.

When `SET CONSTRAINTS` changes the mode of a constraint from `DEFERRED` to `IMMEDIATE`, the new mode takes effect retroactively: any outstanding data modifications that would have been checked at the end of the transaction are instead checked during the execution of the `SET CONSTRAINTS` command. If any such constraint is violated, the `SET CONSTRAINTS` fails (and does not change the constraint mode). Thus, `SET CONSTRAINTS` can be used to force checking of constraints to occur at a specific point in a transaction.

Currently, only `UNIQUE`, `PRIMARY KEY`, `REFERENCES` (foreign key), and `EXCLUDE` constraints are affected by this setting. `NOT NULL` and `CHECK` constraints are always checked immediately when a row is inserted

or modified (*not* at the end of the statement). Uniqueness and exclusion constraints that have not been declared `DEFERRABLE` are also checked immediately.

The firing of triggers that are declared as "constraint triggers" is also controlled by this setting — they fire at the same time that the associated constraint should be checked.

Notes

Because Greenplum Database does not require constraint names to be unique within a schema (but only per-table), it is possible that there is more than one match for a specified constraint name. In this case `SET CONSTRAINTS` will act on all matches. For a non-schema-qualified name, once a match or matches have been found in some schema in the search path, schemas appearing later in the path are not searched.

This command only alters the behavior of constraints within the current transaction. Issuing this outside of a transaction block emits a warning and otherwise has no effect.

Compatibility

This command complies with the behavior defined in the SQL standard, except for the limitation that, in Greenplum Database, it does not apply to `NOT NULL` and `CHECK` constraints. Also, Greenplum Database checks non-deferrable uniqueness constraints immediately, not at end of statement as the standard would suggest.

SET ROLE

Sets the current role identifier of the current session.

Synopsis

```
SET [SESSION | LOCAL] ROLE rolename

SET [SESSION | LOCAL] ROLE NONE

RESET ROLE
```

Description

This command sets the current role identifier of the current SQL-session context to be *rolename*. The role name may be written as either an identifier or a string literal. After `SET ROLE`, permissions checking for SQL commands is carried out as though the named role were the one that had logged in originally.

The specified *rolename* must be a role that the current session user is a member of. If the session user is a superuser, any role can be selected.

The `NONE` and `RESET` forms reset the current role identifier to be the current session role identifier. These forms may be executed by any user.

Parameters

`SESSION`

Specifies that the command takes effect for the current session. This is the default.

`LOCAL`

Specifies that the command takes effect for only the current transaction. After `COMMIT` or `ROLLBACK`, the session-level setting takes effect again. Note that `SET LOCAL` will appear to have no effect if it is executed outside of a transaction.

`rolename`

The name of a role to use for permissions checking in this session.

NONE
RESET

Reset the current role identifier to be the current session role identifier (that of the role used to log in).

Notes

Using this command, it is possible to either add privileges or restrict privileges. If the session user role has the `INHERITS` attribute, then it automatically has all the privileges of every role that it could `SET ROLE` to; in this case `SET ROLE` effectively drops all the privileges assigned directly to the session user and to the other roles it is a member of, leaving only the privileges available to the named role. On the other hand, if the session user role has the `NOINHERITS` attribute, `SET ROLE` drops the privileges assigned directly to the session user and instead acquires the privileges available to the named role.

In particular, when a superuser chooses to `SET ROLE` to a non-superuser role, she loses her superuser privileges.

`SET ROLE` has effects comparable to `SET SESSION AUTHORIZATION`, but the privilege checks involved are quite different. Also, `SET SESSION AUTHORIZATION` determines which roles are allowable for later `SET ROLE` commands, whereas changing roles with `SET ROLE` does not change the set of roles allowed to a later `SET ROLE`.

`SET ROLE` does not process session variables specified by the role's `ALTER ROLE` settings; the session variables are only processed during login.

Examples

```
SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 peter        | paul
```

Compatibility

Greenplum Database allows identifier syntax (*rolename*), while the SQL standard requires the role name to be written as a string literal. SQL does not allow this command during a transaction; Greenplum Database does not make this restriction. The `SESSION` and `LOCAL` modifiers are a Greenplum Database extension, as is the `RESET` syntax.

See Also

`SET SESSION AUTHORIZATION`

SET SESSION AUTHORIZATION

Sets the session role identifier and the current role identifier of the current session.

Synopsis

```
SET [SESSION | LOCAL] SESSION AUTHORIZATION rolename

SET [SESSION | LOCAL] SESSION AUTHORIZATION DEFAULT
```

```
RESET SESSION AUTHORIZATION
```

Description

This command sets the session role identifier and the current role identifier of the current SQL-session context to be *rolename*. The role name may be written as either an identifier or a string literal. Using this command, it is possible, for example, to temporarily become an unprivileged user and later switch back to being a superuser.

The session role identifier is initially set to be the (possibly authenticated) role name provided by the client. The current role identifier is normally equal to the session user identifier, but may change temporarily in the context of `setuid` functions and similar mechanisms; it can also be changed by *`SET ROLE`*. The current user identifier is relevant for permission checking.

The session user identifier may be changed only if the initial session user (the authenticated user) had the superuser privilege. Otherwise, the command is accepted only if it specifies the authenticated user name.

The `DEFAULT` and `RESET` forms reset the session and current user identifiers to be the originally authenticated user name. These forms may be executed by any user.

Parameters

`SESSION`

Specifies that the command takes effect for the current session. This is the default.

`LOCAL`

Specifies that the command takes effect for only the current transaction. After `COMMIT` or `ROLLBACK`, the session-level setting takes effect again. Note that `SET LOCAL` will appear to have no effect if it is executed outside of a transaction.

`rolename`

The name of the role to assume.

`NONE`
`RESET`

Reset the session and current role identifiers to be that of the role used to log in.

Examples

```
SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 paul         | paul
```

Compatibility

The SQL standard allows some other expressions to appear in place of the literal *rolename*, but these options are not important in practice. Greenplum Database allows identifier syntax (*rolename*), which SQL does not. SQL does not allow this command during a transaction; Greenplum Database does not make this restriction. The `SESSION` and `LOCAL` modifiers are a Greenplum Database extension, as is the `RESET` syntax.

See Also

SET ROLE

SET TRANSACTION

Sets the characteristics of the current transaction.

Synopsis

```
SET TRANSACTION [transaction_mode] [READ ONLY | READ WRITE]

SET TRANSACTION SNAPSHOT snapshot_id

SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode
     [READ ONLY | READ WRITE]
     [NOT] DEFERRABLE
```

where *transaction_mode* is one of:

```
   ISOLATION LEVEL {SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
 UNCOMMITTED}
```

Description

The `SET TRANSACTION` command sets the characteristics of the current transaction. It has no effect on any subsequent transactions.

The available transaction characteristics are the transaction isolation level, the transaction access mode (read/write or read-only), and the deferrable mode.

> **Note:** Deferrable transactions require the transaction to be serializable. Greenplum Database does not support serializable transactions, so including the `DEFERRABLE` clause has no effect.

Greenplum Database does not support the `SET TRANSACTION SNAPSHOT` command.

The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently.

- **READ COMMITTED** — A statement can only see rows committed before it began. This is the default.
- **REPEATABLE READ** — All statements in the current transaction can only see rows committed before the first query or data-modification statement executed in the transaction.

The SQL standard defines two additional levels, `READ UNCOMMITTED` and `SERIALIZABLE`. In Greenplum Database `READ UNCOMMITTED` is treated as `READ COMMITTED`. If you specify `SERIALIZABLE`, Greenplum Database falls back to `REPEATABLE READ`.

The transaction isolation level cannot be changed after the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, or `COPY`) of a transaction has been executed.

The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.

The `DEFERRABLE` transaction property has no effect unless the transaction is also `SERIALIZABLE` and `READ ONLY`. When all of these properties are set on a transaction, the transaction may block when first acquiring its snapshot, after which it is able to run without the normal overhead of a `SERIALIZABLE` transaction and without any risk of contributing to or being cancelled by a serialization failure. Because

Greenplum Database does not support serializable transactions, the DEFERRABLE transaction property has no effect in Greenplum Database.

Parameters

SESSION CHARACTERISTICS

Sets the default transaction characteristics for subsequent transactions of a session.

READ UNCOMMITTED
READ COMMITTED
REPEATABLE READ
SERIALIZABLE

The SQL standard defines four transaction isolation levels: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE.

READ UNCOMMITTED allows transactions to see changes made by uncomitted concurrent transactions. This is not possible in Greenplum Database, so READ UNCOMMITTED is treated the same as READ COMMITTED.

READ COMMITTED, the default isolation level in Greenplum Database, guarantees that a statement can only see rows committed before it began. The same statement executed twice in a transaction can produce different results if another concurrent transaction commits after the statement is executed the first time.

The REPEATABLE READ isolation level guarantees that a transaction can only see rows committed before it began. REPEATABLE READ is the strictest transaction isolation level Greenplum Database supports. Applications that use the REPEATABLE READ isolation level must be prepared to retry transactions due to serialization failures.

The SERIALIZABLE transaction isolation level guarantees that all statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction. If a pattern of reads and writes among concurrent serializable transactions would create a situation which could not have occurred for any serial (one-at-a-time) execution of those transactions, one of the transactions will be rolled back with a serialization_failure error. Greenplum Database does not fully support SERIALIZABLE as defined by the standard, so if you specify SERIALIZABLE, Greenplum Database falls back to REPEATABLE READ. See *Compatibility* for more information about transaction serializability in Greenplum Database.

READ WRITE
READ ONLY

Determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: INSERT, UPDATE, DELETE, and COPY FROM if the table they would write to is not a temporary table; all CREATE, ALTER, and DROP commands; GRANT, REVOKE, TRUNCATE; and EXPLAIN ANALYZE and EXECUTE if the command they would execute is among those listed.

[NOT] DEFERRABLE

The DEFERRABLE transaction property has no effect in Greenplum Database because SERIALIZABLE transactions are not supported. If DEFERRABLE is specified and the transaction is also SERIALIZABLE and READ ONLY, the transaction may block when first acquiring its snapshot, after which it is able to run without the normal overhead of a SERIALIZABLE transaction and without any risk of contributing to or being cancelled by a serialization failure. This mode is well suited for long-running reports or backups.

Notes

If SET TRANSACTION is executed without a prior START TRANSACTION or BEGIN, a warning is issued and the command has no effect.

It is possible to dispense with `SET TRANSACTION` by instead specifying the desired transaction modes in `BEGIN` or `START TRANSACTION`.

The session default transaction modes can also be set by setting the configuration parameters *default_transaction_isolation*, *default_transaction_read_only*, and *default_transaction_deferrable*.

Examples

Set the transaction isolation level for the current transaction:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Compatibility

Both commands are defined in the SQL standard. `SERIALIZABLE` is the default transaction isolation level in the standard. In Greenplum Database the default is `READ COMMITTED`. Due to lack of predicate locking, Greenplum Database does not fully support the `SERIALIZABLE` level, so it falls back to the `REPEATABLE READ` level when `SERIAL` is specified. Essentially, a predicate-locking system prevents phantom reads by restricting what is written, whereas a multi-version concurrency control model (MVCC) as used in Greenplum Database prevents them by restricting what is read.

PostgreSQL provides a true serializable isolation level, called serializable snapshot isolation (SSI), which monitors concurrent transactions and rolls back transactions that could introduce serialization anomalies. Greenplum Database does not implement this isolation mode.

In the SQL standard, there is one other transaction characteristic that can be set with these commands: the size of the diagnostics area. This concept is specific to embedded SQL, and therefore is not implemented in the Greenplum Database server.

The `DEFERRABLE` transaction mode is a Greenplum Database language extension.

The SQL standard requires commas between successive *transaction_modes*, but for historical reasons Greenplum Database allows the commas to be omitted.

See Also

BEGIN, *LOCK*

SHOW

Shows the value of a system configuration parameter.

Synopsis

```
SHOW configuration_parameter

SHOW ALL
```

Description

`SHOW` displays the current settings of Greenplum Database system configuration parameters. You can set these parameters with the `SET` statement, or by editing the `postgresql.conf` configuration file of the Greenplum Database master. Note that some parameters viewable by `SHOW` are read-only — their values can be viewed but not set. See the Greenplum Database Reference Guide for details.

Parameters

configuration_parameter

The name of a system configuration parameter.

ALL

Shows the current value of all configuration parameters.

Examples

Show the current setting of the parameter `DateStyle`:

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

Show the current setting of the parameter *geqo*:

```
SHOW geqo;
 geqo
------
 off
(1 row)
```

Show the current setting of all parameters:

```
SHOW ALL;
       name       | setting |                  description
------------------+---------
+-------------------------------------------------------
 application_name | psql    | Sets the application name to be reported in
 sta...
       .
       .
       .
 xmlbinary        | base64  | Sets how binary values are to be encoded in
 XML.
 xmloption        | content | Sets whether XML data in implicit parsing and
 s...
(331 rows)
```

Compatibility

SHOW is a Greenplum Database extension.

See Also

SET, *RESET*

START TRANSACTION

Starts a transaction block.

Synopsis

```
START TRANSACTION [transaction_mode] [READ WRITE | READ ONLY]
```

where *transaction_mode* is:

```
   ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED | READ UNCOMMITTED}
```

Description

`START TRANSACTION` begins a new transaction block. If the isolation level or read/write mode is specified, the new transaction has those characteristics, as if *`SET TRANSACTION`* was executed. This is the same as the `BEGIN` command.

Parameters

`READ UNCOMMITTED`
`READ COMMITTED`
`REPEATABLE READ`
`SERIALIZABLE`

The SQL standard defines four transaction isolation levels: `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`.

`READ UNCOMMITTED` allows transactions to see changes made by uncomitted concurrent transactions. This is not possible in Greenplum Database, so `READ UNCOMMITTED` is treated the same as `READ COMMITTED`.

`READ COMMITTED`, the default isolation level in Greenplum Database, guarantees that a statement can only see rows committed before it began. The same statement executed twice in a transaction can produce different results if another concurrent transaction commits after the statement is executed the first time.

The `REPEATABLE READ` isolation level guarantees that a transaction can only see rows committed before it began. `REPEATABLE READ` is the strictest transaction isolation level Greenplum Database supports. Applications that use the `REPEATABLE READ` isolation level must be prepared to retry transactions due to serialization failures.

The `SERIALIZABLE` transaction isolation level guarantees that executing multiple concurrent transactions produces the same effects as running the same transactions one at a time. If you specify `SERIALIZABLE`, Greenplum Database falls back to `REPEATABLE READ`.

`READ WRITE`
`READ ONLY`

Determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed.

Examples

To begin a transaction block:

```
START TRANSACTION;
```

Compatibility

In the standard, it is not necessary to issue `START TRANSACTION` to start a transaction block: any SQL command implicitly begins a block. Greenplum Database behavior can be seen as implicitly issuing a `COMMIT` after each command that does not follow `START TRANSACTION` (or `BEGIN`), and it is therefore often called 'autocommit'. Other relational database systems may offer an autocommit feature as a convenience.

The SQL standard requires commas between successive *transaction_modes*, but for historical reasons Greenplum Database allows the commas to be omitted.

See also the compatibility section of *`SET TRANSACTION`*.

See Also

BEGIN, *SET TRANSACTION*

TRUNCATE

Empties a table of all rows.

Synopsis

```
TRUNCATE [TABLE] [ONLY] name [ * ] [, ...]
    [ RESTART IDENTITY | CONTINUE IDENTITY ] [CASCADE | RESTRICT]
```

Description

TRUNCATE quickly removes all rows from a table or set of tables. It has the same effect as an unqualified *DELETE* on each table, but since it does not actually scan the tables it is faster. This is most useful on large tables.

You must have the TRUNCATE privilege on the table to truncate table rows.

TRUNCATE acquires an access exclusive lock on the tables it operates on, which blocks all other concurrent operations on the table. When RESTART IDENTITY is specified, any sequences that are to be restarted are likewise locked exclusively. If concurrent access to a table is required, then the DELETE command should be used instead.

Parameters

name

The name (optionally schema-qualified) of a table to truncate. If ONLY is specified before the table name, only that table is truncated. If ONLY is not specified, the table and all its descendant tables (if any) are truncated. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

CASCADE

Because this key word applies to foreign key references (which are not supported in Greenplum Database) it has no effect.

RESTART IDENTITY

Automatically restart sequences owned by columns of the truncated table(s).

CONTINUE IDENTITY

Do not change the values of sequences. This is the default.

RESTRICT

Because this key word applies to foreign key references (which are not supported in Greenplum Database) it has no effect.

Notes

TRUNCATE will not run any user-defined ON DELETE triggers that might exist for the tables.

TRUNCATE will not truncate any tables that inherit from the named table. Only the named table is truncated, not its child tables.

TRUNCATE will not truncate any sub-tables of a partitioned table. If you specify a sub-table of a partitioned table, TRUNCATE will not remove rows from the sub-table and its child tables.

TRUNCATE is not MVCC-safe. After truncation, the table will appear empty to concurrent transactions, if they are using a snapshot taken before the truncation occurred.

TRUNCATE is transaction-safe with respect to the data in the tables: the truncation will be safely rolled back if the surrounding transaction does not commit.

TRUNCATE acquires an ACCESS EXCLUSIVE lock on each table it operates on, which blocks all other concurrent operations on the table. If concurrent access to a table is required, then the DELETE command should be used instead.

When RESTART IDENTITY is specified, the implied ALTER SEQUENCE RESTART operations are also done transactionally; that is, they will be rolled back if the surrounding transaction does not commit. This is unlike the normal behavior of ALTER SEQUENCE RESTART. Be aware that if any additional sequence operations are done on the restarted sequences before the transaction rolls back, the effects of these operations on the sequences will be rolled back, but not their effects on currval(); that is, after the transaction currval() will continue to reflect the last sequence value obtained inside the failed transaction, even though the sequence itself may no longer be consistent with that. This is similar to the usual behavior of currval() after a failed transaction.

Examples

Empty the tables films and distributors:

```
TRUNCATE films, distributors;
```

The same, and also reset any associated sequence generators:

```
TRUNCATE films, distributors RESTART IDENTITY;
```

Compatibility

The SQL:2008 standard includes a TRUNCATE command with the syntax TRUNCATE TABLE *tablename*. The clauses CONTINUE IDENTITY/RESTART IDENTITY also appear in that standard, but have slightly different though related meanings. Some of the concurrency behavior of this command is left implementation-defined by the standard, so the above notes should be considered and compared with other implementations if necessary.

See Also

DELETE, *DROP TABLE*

UPDATE

Updates rows of a table.

Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ONLY] table [[AS] alias]
   SET {column = {expression | DEFAULT} |
   (column [, ...]) = ({expression | DEFAULT} [, ...])} [, ...]
   [FROM fromlist]
   [WHERE condition | WHERE CURRENT OF cursor_name ]
```

Description

UPDATE changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the SET clause; columns not explicitly modified retain their previous values.

By default, UPDATE will update rows in the specified table and all its subtables. If you wish to only update the specific table mentioned, you must use the ONLY clause.

There are two ways to modify a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the FROM clause. Which technique is more appropriate depends on the specific circumstances.

If the WHERE CURRENT OF clause is specified, the row that is updated is the one most recently fetched from the specified cursor.

The WHERE CURRENT OF clause is not supported with replicated tables.

You must have the UPDATE privilege on the table, or at least on the column(s) that are listed to be updated. You must also have the SELECT privilege on any column whose values are read in the *expression*s or *condition*.

> **Note:** As the default, Greenplum Database acquires an EXCLUSIVE lock on tables for UPDATE operations on heap tables. When the Global Deadlock Detector is enabled, the lock mode for UPDATE operations on heap tables is ROW EXCLUSIVE. See *Global Deadlock Detector*.

Outputs

On successful completion, an UPDATE command returns a command tag of the form:

```
UPDATE count
```

where *count* is the number of rows updated. If *count* is 0, no rows matched the condition (this is not considered an error).

Parameters

with_query

The WITH clause allows you to specify one or more subqueries that can be referenced by name in the UPDATE query.

For an UPDATE command that includes a WITH clause, the clause can only contain SELECT commands, the WITH clause cannot contain a data-modifying command (INSERT, UPDATE, or DELETE).

It is possible for the query (SELECT statement) to also contain a WITH clause. In such a case both sets of *with_query* can be referenced within the UPDATE query, but the second one takes precedence since it is more closely nested.

See *WITH Queries (Common Table Expressions)* and *SELECT* for details.

ONLY

If specified, update rows from the named table only. When not specified, any tables inheriting from the named table are also processed.

table

The name (optionally schema-qualified) of an existing table.

alias

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given UPDATE foo AS f, the remainder of the UPDATE statement must refer to this table as f not foo.

column

The name of a column in table. The column name can be qualified with a subfield name or array subscript, if needed. Do not include the table's name in the specification of a target column.

expression

An expression to assign to the column. The expression may use the old values of this and other columns in the table.

DEFAULT

Set the column to its default value (which will be NULL if no specific default expression has been assigned to it).

fromlist

A list of table expressions, allowing columns from other tables to appear in the `WHERE` condition and the update expressions. This is similar to the list of tables that can be specified in the `FROM` clause of a `SELECT` statement. Note that the target table must not appear in the *fromlist*, unless you intend a self-join (in which case it must appear with an *alias* in the *fromlist*).

condition

An expression that returns a value of type boolean. Only rows for which this expression returns true will be updated.

cursor_name

The name of the cursor to use in a `WHERE CURRENT OF` condition. The row to be updated is the one most recently fetched from the cursor. The cursor must be a non-grouping query on the `UPDATE` command target table. See *DECLARE* for more information about creating cursors.

`WHERE CURRENT OF` cannot be specified together with a Boolean condition.

Note that `WHERE CURRENT OF` cannot be specified together with a Boolean condition. The `UPDATE...WHERE CURRENT OF` statement can only be executed on the server, for example in an interactive psql session or a script. Language extensions such as PL/pgSQL do not have support for updatable cursors.

See *DECLARE* for more information about creating cursors.

output_expression

An expression to be computed and returned by the `UPDATE` command after each row is updated. The expression may use any column names of the table or table(s) listed in `FROM`. Write `*` to return all columns.

output_name

A name to use for a returned column.

Notes

`SET` is not allowed on the Greenplum distribution key columns of a table.

When a `FROM` clause is present, what essentially happens is that the target table is joined to the tables mentioned in the from list, and each output row of the join represents an update operation for the target table. When using `FROM` you should ensure that the join produces at most one output row for each row to be modified. In other words, a target row should not join to more than one row from the other table(s). If it does, then only one of the join rows will be used to update the target row, but which one will be used is not readily predictable.

Because of this indeterminacy, referencing other tables only within sub-selects is safer, though often harder to read and slower than using a join.

Executing `UPDATE` and `DELETE` commands directly on a specific partition (child table) of a partitioned table is not supported. Instead, execute these commands on the root partitioned table, the table created with the `CREATE TABLE` command.

For a partitioned table, all the child tables are locked during the `UPDATE` operation when the Global Deadlock Detector is not enabled (the default). Only some of the leaf child tables are locked when the

Global Deadlock Detector is enabled. For information about the Global Deadlock Detector, see *Global Deadlock Detector*.

Examples

Change the word `Drama` to `Dramatic` in the column `kind` of the table `films`:

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Adjust temperature entries and reset precipitation to its default value in one row of the table `weather`:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi =
temp_lo+15, prcp = DEFAULT
WHERE city = 'San Francisco' AND date = '2016-07-03';
```

Use the alternative column-list syntax to do the same update:

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1,
temp_lo+15, DEFAULT)
WHERE city = 'San Francisco' AND date = '2016-07-03';
```

Increment the sales count of the salesperson who manages the account for Acme Corporation, using the `FROM` clause syntax (assuming both tables being joined are distributed in Greenplum Database on the `id` column):

```
UPDATE employees SET sales_count = sales_count + 1 FROM
accounts
WHERE accounts.name = 'Acme Corporation'
AND employees.id = accounts.id;
```

Perform the same operation, using a sub-select in the `WHERE` clause:

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT id FROM accounts WHERE name = 'Acme Corporation');
```

Attempt to insert a new stock item along with the quantity of stock. If the item already exists, instead update the stock count of the existing item. To do this without failing the entire transaction, use savepoints.

```
BEGIN;
-- other operations
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- Assume the above fails because of a unique key violation,
-- so now we issue these commands:
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau
Lafite 2003';
-- continue with other operations, and eventually
COMMIT;
```

Compatibility

This command conforms to the SQL standard, except that the `FROM` clause is a Greenplum Database extension.

According to the standard, the column-list syntax should allow a list of columns to be assigned from a single row-valued expression, such as a sub-select:

```
UPDATE accounts SET (contact_last_name, contact_first_name) =
    (SELECT last_name, first_name FROM salesmen
```

```
      WHERE salesmen.id = accounts.sales_id);
```

This is not currently implemented — the source must be a list of independent expressions.

Some other database systems offer a `FROM` option in which the target table is supposed to be listed again within `FROM`. That is not how Greenplum Database interprets `FROM`. Be careful when porting applications that use this extension.

See Also

DECLARE, *DELETE*, *SELECT*, *INSERT*

VACUUM

Garbage-collects and optionally analyzes a database.

Synopsis

```
VACUUM [({ FULL | FREEZE | VERBOSE | ANALYZE } [, ...])] [table [(column
 [, ...] )]]

VACUUM [FULL] [FREEZE] [VERBOSE] [table]

VACUUM [FULL] [FREEZE] [VERBOSE] ANALYZE
              [table [(column [, ...] )]]
```

Description

`VACUUM` reclaims storage occupied by deleted tuples. In normal Greenplum Database operation, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present on disk until a `VACUUM` is done. Therefore it is necessary to do `VACUUM` periodically, especially on frequently-updated tables.

With no parameter, `VACUUM` processes every table in the current database. With a parameter, `VACUUM` processes only that table.

`VACUUM ANALYZE` performs a `VACUUM` and then an `ANALYZE` for each selected table. This is a handy combination form for routine maintenance scripts. See *ANALYZE* for more details about its processing.

`VACUUM` (without `FULL`) marks deleted and obsoleted data in tables and indexes for future reuse and reclaims space for re-use only if the space is at the end of the table and an exclusive table lock can be easily obtained. Unused space at the start or middle of a table remains as is. With heap tables, this form of the command can operate in parallel with normal reading and writing of the table, as an exclusive lock is not obtained. However, extra space is not returned to the operating system (in most cases); it's just kept available for re-use within the same table. `VACUUM FULL` rewrites the entire contents of the table into a new disk file with no extra space, allowing unused space to be returned to the operating system. This form is much slower and requires an exclusive lock on each table while it is being processed.

With append-optimized tables, `VACUUM` compacts a table by first vacuuming the indexes, then compacting each segment file in turn, and finally vacuuming auxiliary relations and updating statistics. On each segment, visible rows are copied from the current segment file to a new segment file, and then the current segment file is scheduled to be dropped and the new segment file is made available. Plain `VACUUM` of an append-optimized table allows scans, inserts, deletes, and updates of the table while a segment file is compacted. However, an Access Exclusive lock is taken briefly to drop the current segment file and activate the new segment file.

`VACUUM FULL` does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an Access Exclusive lock on each table while it is being processed. The Access Exclusive lock guarantees that the holder is the only transaction accessing the table in any way.

When the option list is surrounded by parentheses, the options can be written in any order. Without parentheses, options must be specified in exactly the order shown above. The parenthesized syntax was added in Greenplum Database 6.0; the unparenthesized syntax is deprecated.

> **Important:** For information on the use of VACUUM, VACUUM FULL, and VACUUM ANALYZE, see *Notes*.

Outputs

When VERBOSE is specified, VACUUM emits progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.

Parameters

FULL

Selects a full vacuum, which may reclaim more space, but takes much longer and exclusively locks the table. This method also requires extra disk space, since it writes a new copy of the table and doesn't release the old copy until the operation is complete. Usually this should only be used when a significant amount of space needs to be reclaimed from within the table.

FREEZE

Specifying FREEZE is equivalent to performing VACUUM with the vacuum_freeze_min_age server configuration parameter set to zero. See *Server Configuration Parameters* for information about vacuum_freeze_min_age.

VERBOSE

Prints a detailed vacuum activity report for each table.

ANALYZE

Updates statistics used by the planner to determine the most efficient way to execute a query.

table

The name (optionally schema-qualified) of a specific table to vacuum. Defaults to all tables in the current database.

column

The name of a specific column to analyze. Defaults to all columns. If a column list is specified, ANALYZE is implied.

Notes

VACUUM cannot be executed inside a transaction block.

Vacuum active databases frequently (at least nightly), in order to remove expired rows. After adding or deleting a large number of rows, running the VACUUM ANALYZE command for the affected table might be useful. This updates the system catalogs with the results of all recent changes, and allows the Greenplum Database query optimizer to make better choices in planning queries.

> **Important:** PostgreSQL has a separate optional server process called the *autovacuum daemon*, whose purpose is to automate the execution of VACUUM and ANALYZE commands. Greenplum Database enables the autovacuum daemon to perform VACUUM operations only on the Greenplum Database template database template0. Autovacuum is enabled for template0 because connections are not allowed to template0. The autovacuum daemon performs VACUUM operations on template0 to manage transaction IDs (XIDs) and help avoid transaction ID wraparound issues in template0.
>
> Manual VACUUM operations must be performed in user-defined databases to manage transaction IDs (XIDs) in those databases.

VACUUM causes a substantial increase in I/O traffic, which can cause poor performance for other active sessions. Therefore, it is advisable to vacuum the database at low usage times.

VACUUM commands skip external and foreign tables.

VACUUM FULL reclaims all expired row space, however it requires an exclusive lock on each table being processed, is a very expensive operation, and might take a long time to complete on large, distributed Greenplum Database tables. Perform VACUUM FULL operations during database maintenance periods.

The FULL option is not recommended for routine use, but might be useful in special cases. An example is when you have deleted or updated most of the rows in a table and would like the table to physically shrink to occupy less disk space and allow faster table scans. VACUUM FULL will usually shrink the table more than a plain VACUUM would.

As an alternative to VACUUM FULL, you can re-create the table with a CREATE TABLE AS statement and drop the old table.

For append-optimized tables, VACUUM requires enough available disk space to accommodate the new segment file during the VACUUM process. If the ratio of hidden rows to total rows in a segment file is less than a threshold value (10, by default), the segment file is not compacted. The threshold value can be configured with the gp_appendonly_compaction_threshold server configuration parameter. VACUUM FULL ignores the threshold and rewrites the segment file regardless of the ratio. VACUUM can be disabled for append-optimized tables using the gp_appendonly_compaction server configuration parameter. See *Server Configuration Parameters* for information about the server configuration parameters.

If a concurrent serializable transaction is detected when an append-optimized table is being vacuumed, the current and subsequent segment files are not compacted. If a segment file has been compacted but a concurrent serializable transaction is detected in the transaction that drops the original segment file, the drop is skipped. This could leave one or two segment files in an "awaiting drop" state after the vacuum has completed.

For more information about concurrency control in Greenplum Database, see "Routine System Maintenance Tasks" in *Greenplum Database Administrator Guide*.

Examples

To clean a single table onek, analyze it for the optimizer and print a detailed vacuum activity report:

```
VACUUM (VERBOSE, ANALYZE) onek;
```

Vacuum all tables in the current database:

```
VACUUM;
```

Vacuum a specific table only:

```
VACUUM (VERBOSE, ANALYZE) mytable;
```

Vacuum all tables in the current database and collect statistics for the query optimizer:

```
VACUUM ANALYZE;
```

Compatibility

There is no VACUUM statement in the SQL standard.

See Also

ANALYZE

VALUES

Computes a set of rows.

Synopsis

```
VALUES ( expression [, ...] ) [, ...]
   [ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
   [LIMIT { count | ALL } ]
   [OFFSET start [ ROW | ROWS ] ]
   [FETCH { FIRST | NEXT } [count ] { ROW | ROWS } ONLY ]
```

Description

VALUES computes a row value or set of row values specified by value expressions. It is most commonly used to generate a "constant table" within a larger command, but it can be used on its own.

When more than one row is specified, all the rows must have the same number of elements. The data types of the resulting table's columns are determined by combining the explicit or inferred types of the expressions appearing in that column, using the same rules as for UNION.

Within larger commands, VALUES is syntactically allowed anywhere that SELECT is. Because it is treated like a SELECT by the grammar, it is possible to use the ORDER BY, LIMIT (or equivalent FETCH FIRST), and OFFSET clauses with a VALUES command.

Parameters

expression

A constant or expression to compute and insert at the indicated place in the resulting table (set of rows). In a VALUES list appearing at the top level of an INSERT, an expression can be replaced by DEFAULT to indicate that the destination column's default value should be inserted. DEFAULT cannot be used when VALUES appears in other contexts.

sort_expression

An expression or integer constant indicating how to sort the result rows. This expression may refer to the columns of the VALUES result as column1, column2, etc. For more details, see "The ORDER BY Clause" in the parameters for *SELECT*.

operator

A sorting operator. For more details, see "The ORDER BY Clause" in the parameters for *SELECT*.

LIMIT count
OFFSET start

The maximum number of rows to return. For more details, see "The LIMIT Clause" in the parameters for *SELECT*.

Notes

VALUES lists with very large numbers of rows should be avoided, as you may encounter out-of-memory failures or poor performance. VALUES appearing within INSERT is a special case (because the desired column types are known from the INSERT's target table, and need not be inferred by scanning the VALUES list), so it can handle larger lists than are practical in other contexts.

Examples

A bare VALUES command:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

This will return a table of two columns and three rows. It is effectively equivalent to:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

More usually, VALUES is used within a larger SQL command. The most common use is in INSERT:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

In the context of INSERT, entries of a VALUES list can be DEFAULT to indicate that the column default should be used here instead of specifying a value:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82
minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

VALUES can also be used where a sub-SELECT might be written, for example in a FROM clause:

```
SELECT f.* FROM films f, (VALUES('MGM', 'Horror'), ('UA',
'Sci-Fi')) AS t (studio, kind) WHERE f.studio = t.studio AND
f.kind = t.kind;
UPDATE employees SET salary = salary * v.increase FROM
(VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno,
target, increase) WHERE employees.depno = v.depno AND
employees.sales >= v.target;
```

Note that an AS clause is required when VALUES is used in a FROM clause, just as is true for SELECT. It is not required that the AS clause specify names for all the columns, but it is good practice to do so. The default column names for VALUES are column1, column2, etc. in Greenplum Database, but these names might be different in other database systems.

When VALUES is used in INSERT, the values are all automatically coerced to the data type of the corresponding destination column. When it is used in other contexts, it may be necessary to specify the correct data type. If the entries are all quoted literal constants, coercing the first is sufficient to determine the assumed type for all:

```
SELECT * FROM machines WHERE ip_address IN
(VALUES('192.168.0.1'::inet), ('192.168.0.10'),
('192.0.2.43'));
```

> **Note:** For simple IN tests, it is better to rely on the list-of-scalars form of IN than to write a VALUES query as shown above. The list of scalars method requires less writing and is often more efficient.

Compatibility

VALUES conforms to the SQL standard. LIMIT and OFFSET are Greenplum Database extensions; see also under *SELECT*.

See Also

INSERT, *SELECT*

Data Types

Greenplum Database has a rich set of native data types available to users. Users may also define new data types using the `CREATE TYPE` command. This reference shows all of the built-in data types. In addition to the types listed here, there are also some internally used data types, such as *oid* (object identifier), but those are not documented in this guide.

Additional modules that you register may also install new data types. The `hstore` module, for example, introduces a new data type and associated functions for working with key-value pairs. See *hstore*. The `citext` module adds a case-insensitive text data type. See *citext*.

The following data types are specified by SQL: *bit*, *bit varying*, *boolean*, *character varying, varchar*, *character, char*, *date*, *double precision*, *integer*, *interval*, *numeric*, *decimal*, *real*, *smallint*, *time* (with or without time zone), and *timestamp* (with or without time zone).

Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL (and Greenplum Database), such as geometric paths, or have several possibilities for formats, such as the date and time types. Some of the input and output functions are not invertible. That is, the result of an output function may lose accuracy when compared to the original input.

Table 87: Greenplum Database Built-in Data Types

Name	Alias	Size	Range	Description
bigint	int8	8 bytes	-922337203 6854775808 to 922337203 6854775807	large range integer
bigserial	serial8	8 bytes	1 to 922337203 6854775807	large autoincrementing integer
bit [(n)]		*n* bits	*bit string constant*	fixed-length bit string
bit varying [(n)]*Footnote.*	varbit	actual number of bits	*bit string constant*	variable-length bit string
boolean	bool	1 byte	true/false, t/f, yes/ no, y/n, 1/0	logical boolean (true/false)
box		32 bytes	((x1,y1),(x2,y2))	rectangular box in the plane - not allowed in distribution key columns.
bytea*Footnote.*		1 byte + *binary string*	sequence of *octets*	variable-length binary string
character [(n)]*Footnote.*	char [(n)]	1 byte + *n*	strings up to *n* characters in length	fixed-length, blank padded

[9] For variable length data types, if the data is greater than or equal to 127 bytes, the storage overhead is 4 bytes instead of 1.

Name	Alias	Size	Range	Description
character varying [(n)]*Footnote.*	varchar [(n)]	1 byte + *string size*	strings up to *n* characters in length	variable-length with limit
cidr		12 or 24 bytes		IPv4 and IPv6 networks
circle		24 bytes	<(x,y),r> (center and radius)	circle in the plane - not allowed in distribution key columns.
date		4 bytes	4713 BC - 294,277 AD	calendar date (year, month, day)
decimal [(p, s)]*Footnote.*	numeric [(p, s)]	variable	no limit	user-specified precision, exact
double precision	float8 float	8 bytes	15 decimal digits precision	variable-precision, inexact
inet		12 or 24 bytes		IPv4 and IPv6 hosts and networks
integer	int, int4	4 bytes	-2147483648 to +2147483647	usual choice for integer
interval [*fields*] [(p)]		16 bytes	-178000000 years to 178000000 years	time span
json		1 byte + json size	json of any length	variable unlimited length
jsonb		1 byte + binary string	json of any length in a decomposed binary format	variable unlimited length
lseg		32 bytes	((x1,y1),(x2,y2))	line segment in the plane - not allowed in distribution key columns.
macaddr		6 bytes		MAC addresses
money		8 bytes	-92233720368547758.08 to +92233720368547758.07	currency amount
path*Footnote.*		16+16n bytes	[(x1,y1),...]	geometric path in the plane - not allowed in distribution key columns.

Name	Alias	Size	Range	Description
point		16 bytes	(x,y)	geometric point in the plane - not allowed in distribution key columns.
polygon		40+16n bytes	((x1,y1),...)	closed geometric path in the plane - not allowed in distribution key columns.
real	float4	4 bytes	6 decimal digits precision	variable-precision, inexact
serial	serial4	4 bytes	1 to 2147483647	autoincrementing integer
smallint	int2	2 bytes	-32768 to +32767	small range integer
text*Footnote.*		1 byte + *string size*	strings of any length	variable unlimited length
time [(p)] [without time zone]		8 bytes	00:00:00[.000000] - 24:00:00[.000000]	time of day only
time [(p)] with time zone	timetz	12 bytes	00:00:00+1359 - 24:00:00-1359	time of day only, with time zone
timestamp [(p)] [without time zone]		8 bytes	4713 BC - 294,277 AD	both date and time
timestamp [(p)] with time zone	timestamptz	8 bytes	4713 BC - 294,277 AD	both date and time, with time zone
uuid		16 bytes		Universally Unique Identifiers according to RFC 4122, ISO/IEC 9834-8:2005
xml*Footnote.*		1 byte + *xml size*	xml of any length	variable unlimited length
txid_snapshot				user-level transaction ID snapshot

Date/Time Types

Greenplum supports the full set of SQL date and time types, shown in *Table 88: Date/Time Types*. The operations available on these data types are described in *Date/Time Functions and Operators* in the PostgreSQL documentation. Dates are counted according to the Gregorian calendar, even in years before that calendar was introduced (see *History of Units* in the PostgreSQL documentation for more information).

Table 88: Date/Time Types

Name	Storage Size	Description	Low Value	High Value	Resolution
timestamp [(*p*)] [without time zone]	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond / 14 digits
timestamp [(*p*)] with time zone	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond / 14 digits
date	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
time [(*p*)] [without time zone]	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond / 14 digits
time [(*p*)] with time zone	12 bytes	times of day only, with time zone	00:00:00+1459	24:00:00-1459	1 microsecond / 14 digits
interval [*fields*] [(*p*)]	16 bytes	time interval	-178000000 years	178000000 years	1 microsecond / 14 digits

> **Note:** The SQL standard requires that writing just `timestamp` be equivalent to `timestamp without time zone`, and Greenplum honors that behavior. `timestamptz` is accepted as an abbreviation for `timestamp with time zone`; this is a PostgreSQL extension.

`time`, `timestamp`, and `interval` accept an optional precision value *p* which specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The allowed range of *p* is from 0 to 6 for the `timestamp` and `interval` types.

> **Note:** When `timestamp` values are stored as eight-byte integers (currently the default), microsecond precision is available over the full range of values. When `timestamp` values are stored as double precision floating-point numbers instead (a deprecated compile-time option), the effective limit of precision might be less than 6. `timestamp` values are stored as seconds before or after midnight 2000-01-01. When `timestamp` values are implemented using floating-point numbers, microsecond precision is achieved for dates within a few years of 2000-01-01, but the precision degrades for dates further away. Note that using floating-point datetimes allows a larger range of `timestamp` values to be represented than shown above: from 4713 BC up to 5874897 AD.
>
> The same compile-time option also determines whether `time` and `interval` values are stored as floating-point numbers or eight-byte integers. In the floating-point case, large `interval` values degrade in precision as the size of the interval increases.

For the `time` types, the allowed range of *p* is from 0 to 6 when eight-byte integer storage is used, or from 0 to 10 when floating-point storage is used.

The `interval` type has an additional option, which is to restrict the set of stored fields by writing one of these phrases:

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
```

```
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

Note that if both *fields* and *p* are specified, the *fields* must include `SECOND`, since the precision applies only to the seconds.

The type `time with time zone` is defined by the SQL standard, but the definition exhibits properties which lead to questionable usefulness. In most cases, a combination of `date`, `time`, `timestamp without time zone`, and `timestamp with time zone` should provide a complete range of date/time functionality required by any application.

The types `abstime` and `reltime` are lower precision types which are used internally. You are discouraged from using these types in applications; these internal types might disappear in a future release.

Greenplum Database 6 and later releases do not automatically cast text from the deprecated timestamp format `YYYYMMDDHH24MISS`. The format could not be parsed unambiguously in previous Greenplum Database releases.

For example, this command returns an error in Greenplum Database 6. In previous releases, a timestamp is returned.

```
# select to_timestamp('20190905140000');
```

In Greenplum Database 6, this command returns a timestamp.

```
# select to_timestamp('20190905140000','YYYYMMDDHH24MISS');
```

Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional POSTGRES, and others. For some formats, ordering of day, month, and year in date input is ambiguous and there is support for specifying the expected ordering of these fields. Set the *DateStyle* parameter to `MDY` to select month-day-year interpretation, `DMY` to select day-month-year interpretation, or `YMD` to select year-month-day interpretation.

Greenplum is more flexible in handling date/time input than the SQL standard requires. See *Appendix B. Date/Time Support* in the PostgreSQL documentation for the exact parsing rules of date/time input and for the recognized text fields including months, days of the week, and time zones.

Remember that any date or time literal input needs to be enclosed in single quotes, like text strings. SQL requires the following syntax

```
type [ (p) ] 'value'
```

where *p* is an optional precision specification giving the number of fractional digits in the seconds field. Precision can be specified for `time`, `timestamp`, and `interval` types. The allowed values are mentioned above. If no precision is specified in a constant specification, it defaults to the precision of the literal value.

Dates

Table 89: Date Input shows some possible inputs for the `date` type.

Table 89: Date Input

Example	Description
1999-01-08	ISO 8601; January 8 in any mode (recommended format)
January 8, 1999	unambiguous in any *datestyle* input mode
1/8/1999	January 8 in `MDY` mode; August 1 in `DMY` mode
1/18/1999	January 18 in `MDY` mode; rejected in other modes
01/02/03	January 2, 2003 in `MDY` mode; February 1, 2003 in `DMY` mode; February 3, 2001 in `YMD` mode
1999-Jan-08	January 8 in any mode
Jan-08-1999	January 8 in any mode
08-Jan-1999	January 8 in any mode
99-Jan-08	January 8 in `YMD` mode, else error
08-Jan-99	January 8, except error in `YMD` mode
Jan-08-99	January 8, except error in `YMD` mode
19990108	ISO 8601; January 8, 1999 in any mode
990108	ISO 8601; January 8, 1999 in any mode
1999.008	year and day of year
J2451187	Julian date
January 8, 99 BC	year 99 BC

Times

The time-of-day types are `time [ (`*p*`) ] without time zone` and `time [ (`*p*`) ] with time zone`. `time` alone is equivalent to `time without time zone`.

Valid input for these types consists of a time of day followed by an optional time zone. (See *Table 90: Time Input* and *Table 91: Time Zone Input*.) If a time zone is specified in the input for `time without time zone`, it is silently ignored. You can also specify a date but it will be ignored, except when you use a time zone name that involves a daylight-savings rule, such as `America/New_York`. In this case specifying the date is required in order to determine whether standard or daylight-savings time applies. The appropriate time zone offset is recorded in the `time with time zone` value.

Table 90: Time Input

Example	Description
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601
`040506`	ISO 8601
`04:05 AM`	same as 04:05; AM does not affect value
`04:05 PM`	same as 16:05; input hour must be <= 12

Example	Description
`04:05:06.789-8`	ISO 8601
`04:05:06-08:00`	ISO 8601
`04:05-08:00`	ISO 8601
`040506-08`	ISO 8601
`04:05:06 PST`	time zone specified by abbreviation
`2003-04-12 04:05:06 America/New_York`	time zone specified by full name

Table 91: Time Zone Input

Example	Description
`PST`	Abbreviation (for Pacific Standard Time)
`America/New_York`	Full time zone name
`PST8PDT`	POSIX-style time zone specification
`-8:00`	ISO-8601 offset for PST
`-800`	ISO-8601 offset for PST
`-8`	ISO-8601 offset for PST
`zulu`	Military abbreviation for UTC
`z`	Short form of `zulu`

Refer to *Time Zones* for more information on how to specify time zones.

Time Stamps

Valid input for the time stamp types consists of the concatenation of a date and a time, followed by an optional time zone, followed by an optional `AD` or `BC`. (Alternatively, `AD`/`BC` can appear before the time zone, but this is not the preferred ordering.) Thus: `1999-01-08 04:05:06` and: `1999-01-08 04:05:06 -8:00` are valid values, which follow the ISO 8601 standard. In addition, the common format: `January 8 04:05:06 1999 PST` is supported.

The SQL standard differentiates `timestamp without time zone` and `timestamp with time zone` literals by the presence of a `+` or `-` symbol and time zone offset after the time. Hence, according to the standard, `TIMESTAMP '2004-10-19 10:23:54'` is a `timestamp without time zone`, while `TIMESTAMP '2004-10-19 10:23:54+02'` is a `timestamp with time zone`. Greenplum never examines the content of a literal string before determining its type, and therefore will treat both of the above as `timestamp without time zone`. To ensure that a literal is treated as `timestamp with time zone`, give it the correct explicit type: `TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'` In a literal that has been determined to be `timestamp without time zone`, Greenplum will silently ignore any time zone indication. That is, the resulting value is derived from the date/time fields in the input value, and is not adjusted for time zone.

For `timestamp with time zone`, the internally stored value is always in UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, GMT). An input value that has an explicit time zone specified is converted to UTC using the appropriate offset for that time zone. If no time zone is stated in the input string, then it is assumed to be in the time zone indicated by the system's `TimeZone` parameter, and is converted to UTC using the offset for the *timezone* zone.

When a `timestamp with time zone` value is output, it is always converted from UTC to the current *timezone* zone, and displayed as local time in that zone. To see the time in another time zone,

either change *timezone* or use the `AT TIME ZONE` construct (see *AT TIME ZONE* in the PostgreSQL documentation).

Conversions between `timestamp without time zone` and `timestamp with time zone` normally assume that the `timestamp without time zone` value should be taken or given as *timezone* local time. A different time zone can be specified for the conversion using `AT TIME ZONE`.

Special Values

Greenplum supports several special date/time input values for convenience, as shown in *Table 92: Special Date/Time Inputs*. The values `infinity` and `-infinity` are specially represented inside the system and will be displayed unchanged; but the others are simply notational shorthands that will be converted to ordinary date/time values when read. (In particular, `now` and related strings are converted to a specific time value as soon as they are read.) All of these values need to be enclosed in single quotes when used as constants in SQL commands.

Table 92: Special Date/Time Inputs

Input String	Valid Types	Description
`epoch`	`date`, `timestamp`	1970-01-01 00:00:00+00 (Unix system time zero)
`infinity`	`date`, `timestamp`	later than all other time stamps
`-infinity`	`date`, `timestamp`	earlier than all other time stamps
`now`	`date`, `time`, `timestamp`	current transaction's start time
`today`	`date`, `timestamp`	midnight today
`tomorrow`	`date`, `timestamp`	midnight tomorrow
`yesterday`	`date`, `timestamp`	midnight yesterday
`allballs`	`time`	00:00:00.00 UTC

The following SQL-compatible functions can also be used to obtain the current time value for the corresponding data type: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, `LOCALTIMESTAMP`. The latter four accept an optional subsecond precision specification. (See *Current Date/Time* in the PostgreSQL documentation.) Note that these are SQL functions and are *not* recognized in data input strings.

Date/Time Output

The output format of the date/time types can be set to one of the four styles ISO 8601, SQL (Ingres), traditional POSTGRES (Unix `date` format), or German. The default is the ISO format. (The SQL standard requires the use of the ISO 8601 format. The name of the `SQL` output format is a historical accident.) *Table 93: Date/Time Output Styles* shows examples of each output style. The output of the `date` and `time` types is generally only the date or time part in accordance with the given examples. However, the POSTGRES style outputs date-only values in ISO format.

Table 93: Date/Time Output Styles

Style Specification	Description	Example
`ISO`	ISO 8601, SQL standard	`1997-12-17 07:37:16-08`
`SQL`	traditional style	`12/17/1997 07:37:16.00 PST`

Style Specification	Description	Example
`Postgres`	original style	`Wed Dec 17 07:37:16 1997 PST`
`German`	regional style	`17.12.1997 07:37:16.00 PST`

> **Note:** ISO 8601 specifies the use of uppercase letter `T` to separate the date and time. Greenplum accepts that format on input, but on output it uses a space rather than `T`, as shown above. This is for readability and for consistency with RFC 3339 as well as some other database systems.

In the SQL and POSTGRES styles, day appears before month if DMY field ordering has been specified, otherwise month appears before day. (See *Table 89: Date Input* for how this setting also affects interpretation of input values.) *Table 94: Date Order Conventions* shows examples.

Table 94: Date Order Conventions

datestyle Setting	Input Ordering	Example Output
`SQL, DMY`	*day*/*month*/*year*	`17/12/1997 15:37:16.00 CET`
`SQL, MDY`	*month*/*day*/*year*	`12/17/1997 07:37:16.00 PST`
`Postgres, DMY`	*day*/*month*/*year*	`Wed 17 Dec 07:37:16 1997 PST`

The date/time style can be selected by the user using the `SET datestyle` command, the `DateStyle` parameter in the `postgresql.conf` configuration file, or the `PGDATESTYLE` environment variable on the server or client.

The formatting function `to_char` (see *Data Type Formatting Functions*) is also available as a more flexible way to format date/time output.

Time Zones

Time zones, and time-zone conventions, are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900s, but continue to be prone to arbitrary changes, particularly with respect to daylight-savings rules. Greenplum uses the widely-used IANA (Olson) time zone database for information about historical time zone rules. For times in the future, the assumption is that the latest known rules for a given time zone will continue to be observed indefinitely far into the future.

Greenplum endeavors to be compatible with the SQL standard definitions for typical usage. However, the SQL standard has an odd mix of date and time types and capabilities. Two obvious problems are:

1. Although the `date` type cannot have an associated time zone, the `time` type can. Time zones in the real world have little meaning unless associated with a date as well as a time, since the offset can vary through the year with daylight-saving time boundaries.
2. The default time zone is specified as a constant numeric offset from UTC. It is therefore impossible to adapt to daylight-saving time when doing date/time arithmetic across DST boundaries.

To address these difficulties, we recommend using date/time types that contain both date and time when using time zones. We do *not* recommend using the type `time with time zone` (though it is supported by Greenplum for legacy applications and for compliance with the SQL standard). Greenplum assumes your local time zone for any type containing only date or time.

All timezone-aware dates and times are stored internally in UTC. They are converted to local time in the zone specified by the `TimeZone` configuration parameter before being displayed to the client.

Greenplum allows you to specify time zones in three different forms:

1. A full time zone name, for example `America/New_York`. The recognized time zone names are listed in the `pg_timezone_names` view. Greenplum uses the widely-used IANA time zone data for this purpose, so the same time zone names are also recognized by other software.
2. A time zone abbreviation, for example `PST`. Such a specification merely defines a particular offset from UTC, in contrast to full time zone names which can imply a set of daylight savings transition-date rules as well. The recognized abbreviations are listed in the `pg_timezone_abbrevs` view. You cannot set the configuration parameters *TimeZone* or *log_timezone* to a time zone abbreviation, but you can use abbreviations in date/time input values and with the `AT TIME ZONE` operator.
3. In addition to the timezone names and abbreviations, Greenplum will accept POSIX-style time zone specifications of the form *STDoffset* or *STDoffsetDST*, where *STD* is a zone abbreviation, *offset* is a numeric offset in hours west from UTC, and *DST* is an optional daylight-savings zone abbreviation, assumed to stand for one hour ahead of the given offset. For example, if `EST5EDT` were not already a recognized zone name, it would be accepted and would be functionally equivalent to United States East Coast time. In this syntax, a zone abbreviation can be a string of letters, or an arbitrary string surrounded by angle brackets (<>). When a daylight-savings zone abbreviation is present, it is assumed to be used according to the same daylight-savings transition rules used in the IANA time zone database's entry. In a standard Greenplum installation, is the same as `US/Eastern`, so that POSIX-style time zone specifications follow USA daylight-savings rules. If needed, you can adjust this behavior by replacing the file.

In short, this is the difference between abbreviations and full names: abbreviations represent a specific offset from UTC, whereas many of the full names imply a local daylight-savings time rule, and so have two possible UTC offsets. As an example, `2014-06-04 12:00 America/New_York` represents noon local time in New York, which for this particular date was Eastern Daylight Time (UTC-4). So `2014-06-04 12:00 EDT` specifies that same time instant. But `2014-06-04 12:00 EST` specifies noon Eastern Standard Time (UTC-5), regardless of whether daylight savings was nominally in effect on that date.

To complicate matters, some jurisdictions have used the same timezone abbreviation to mean different UTC offsets at different times; for example, in Moscow `MSK` has meant UTC+3 in some years and UTC+4 in others. Greenplum interprets such abbreviations according to whatever they meant (or had most recently meant) on the specified date; but, as with the `EST` example above, this is not necessarily the same as local civil time on that date.

One should be wary that the POSIX-style time zone feature can lead to silently accepting bogus input, since there is no check on the reasonableness of the zone abbreviations. For example, `SET TIMEZONE TO FOOBAR0` will work, leaving the system effectively using a rather peculiar abbreviation for UTC. Another issue to keep in mind is that in POSIX time zone names, positive offsets are used for locations of Greenwich. Everywhere else, Greenplum follows the ISO-8601 convention that positive timezone offsets are of Greenwich.

In all cases, timezone names and abbreviations are recognized case-insensitively.

Neither timezone names nor abbreviations are hard-wired into the server; they are obtained from configuration files (see *Configuring Localization Settings*).

The `TimeZone` configuration parameter can be set in the file , or in any of the other standard ways for setting configuration parameters. There are also some special ways to set it:

1. The SQL command `SET TIME ZONE` sets the time zone for the session. This is an alternative spelling of `SET TIMEZONE TO` with a more SQL-spec-compatible syntax.
2. The `PGTZ` environment variable is used by `libpq` clients to send a `SET TIME ZONE` command to the server upon connection.

Interval Input

`interval` values can be written using the following verbose syntax:

```
@ quantity unit quantity unit... direction
```

where *quantity* is a number (possibly signed); *unit* is `microsecond`, `millisecond`, `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium`, or abbreviations or plurals of these units; *direction* can be `ago` or empty. The at sign (`@`) is optional noise. The amounts of the different units are implicitly added with appropriate sign accounting. `ago` negates all the fields. This syntax is also used for interval output, if *IntervalStyle* is set to `postgres_verbose`.

Quantities of days, hours, minutes, and seconds can be specified without explicit unit markings. For example, `'1 12:59:10'` is read the same as `'1 day 12 hours 59 min 10 sec'`. Also, a combination of years and months can be specified with a dash; for example `'200-10'` is read the same as `'200 years 10 months'`. (These shorter forms are in fact the only ones allowed by the SQL standard, and are used for output when *IntervalStyle* is set to `sql_standard`.)

Interval values can also be written as ISO 8601 time intervals, using either the `format with designators` of the standard's section 4.4.3.2 or the `alternative format` of section 4.4.3.3. The format with designators looks like this:

```
P quantity unit quantity unit ...  T  quantity unit ...
```

The string must start with a `P`, and may include a `T` that introduces the time-of-day units. The available unit abbreviations are given in *Table 95: ISO 8601 Interval Unit Abbreviations*. Units may be omitted, and may be specified in any order, but units smaller than a day must appear after `T`. In particular, the meaning of `M` depends on whether it is before or after `T`.

Table 95: ISO 8601 Interval Unit Abbreviations

Abbreviation	Meaning
Y	Years
M	Months (in the date part)
W	Weeks
D	Days
H	Hours
M	Minutes (in the time part)
S	Seconds

In the alternative format:

```
P  years-months-days   T hours:minutes:seconds
```

the string must begin with `P`, and a `T` separates the date and time parts of the interval. The values are given as numbers similar to ISO 8601 dates.

When writing an interval constant with a *fields* specification, or when assigning a string to an interval column that was defined with a *fields* specification, the interpretation of unmarked quantities depends on the *fields*. For example `INTERVAL '1' YEAR` is read as 1 year, whereas `INTERVAL '1'` means 1 second. Also, field values `to the right` of the least significant field allowed by the *fields* specification are silently discarded. For example, writing `INTERVAL '1 day 2:03:04' HOUR TO MINUTE` results in dropping the seconds field, but not the day field.

According to the SQL standard all fields of an interval value must have the same sign, so a leading negative sign applies to all fields; for example the negative sign in the interval literal `'-1 2:03:04'`

applies to both the days and hour/minute/second parts. Greenplum allows the fields to have different signs, and traditionally treats each field in the textual representation as independently signed, so that the hour/minute/second part is considered positive in this example. If *IntervalStyle* is set to `sql_standard` then a leading sign is considered to apply to all fields (but only if no additional signs appear). Otherwise the traditional Greenplum interpretation is used. To avoid ambiguity, it's recommended to attach an explicit sign to each field if any field is negative.

In the verbose input format, and in some fields of the more compact input formats, field values can have fractional parts; for example `'1.5 week'` or `'01:02:03.45'`. Such input is converted to the appropriate number of months, days, and seconds for storage. When this would result in a fractional number of months or days, the fraction is added to the lower-order fields using the conversion factors 1 month = 30 days and 1 day = 24 hours. For example, `'1.5 month'` becomes 1 month and 15 days. Only seconds will ever be shown as fractional on output.

Table 96: Interval Input shows some examples of valid `interval` input.

Table 96: Interval Input

Example	Description
1-2	SQL standard format: 1 year 2 months
3 4:05:06	SQL standard format: 3 days 4 hours 5 minutes 6 seconds
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	Traditional Postgres format: 1 year 2 months 3 days 4 hours 5 minutes 6 seconds
P1Y2M3DT4H5M6S	ISO 8601 `format with designators`: same meaning as above
P0001-02-03T04:05:06	ISO 8601 `alternative format`: same meaning as above

Internally `interval` values are stored as months, days, and seconds. This is done because the number of days in a month varies, and a day can have 23 or 25 hours if a daylight savings time adjustment is involved. The months and days fields are integers while the seconds field can store fractions. Because intervals are usually created from constant strings or `timestamp` subtraction, this storage method works well in most cases, but can cause unexpected results: `SELECT EXTRACT(hours from '80 minutes'::interval); date_part ----------- 1 SELECT EXTRACT(days from '80 hours'::interval); date_part ----------- 0` Functions `justify_days` and `justify_hours` are available for adjusting days and hours that overflow their normal ranges.

Interval Output

The output format of the interval type can be set to one of the four styles `sql_standard`, `postgres`, `postgres_verbose`, or `iso_8601`, using the command `SET intervalstyle`. The default is the `postgres` format. *Table 97: Interval Output Style Examples* shows examples of each output style.

The `sql_standard` style produces output that conforms to the SQL standard's specification for interval literal strings, if the interval value meets the standard's restrictions (either year-month only or day-time only, with no mixing of positive and negative components). Otherwise the output looks like a standard year-month literal string followed by a day-time literal string, with explicit signs added to disambiguate mixed-sign intervals.

The output of the `postgres` style matches the output of PostgreSQL releases prior to 8.4 when the *DateStyle* parameter was set to `ISO`.

The output of the `postgres_verbose` style matches the output of PostgreSQL releases prior to 8.4 when the *DateStyle* parameter was set to non-`ISO` output.

The output of the `iso_8601` style matches the `format with designators` described in section 4.4.3.2 of the ISO 8601 standard.

Table 97: Interval Output Style Examples

Style Specification	Year-Month Interval	Day-Time Interval	Mixed Interval
`sql_standard`	1-2	3 4:05:06	-1-2 +3 -4:05:06
`postgres`	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
`postgres_verbose`	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
`iso_8601`	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

Pseudo-Types

Greenplum Database supports special-purpose data type entries that are collectively called *pseudo-types*. A pseudo-type cannot be used as a column data type, but it can be used to declare a function's argument or result type. Each of the available pseudo-types is useful in situations where a function's behavior does not correspond to simply taking or returning a value of a specific SQL data type.

Functions coded in procedural languages can use pseudo-types only as allowed by their implementation languages. The procedural languages all forbid use of a pseudo-type as an argument type, and allow only *void* and *record* as a result type.

A function with the pseudo-type *record* as a return data type returns an unspecified row type. The *record* represents an array of possibly-anonymous composite types. Since composite datums carry their own type identification, no extra knowledge is needed at the array level.

The pseudo-type *void* indicates that a function returns no value.

Note: Greenplum Database does not support triggers and the pseudo-type *trigger*.

The types *anyelement*, *anyarray*, *anynonarray*, and *anyenum* are pseudo-types called polymorphic types. Some procedural languages also support polymorphic functions using the types *anyarray*, *anyelement*, *anyenum*, and *anynonarray*.

The pseudo-type *anytable* is a Greenplum Database type that specifies a table expression—an expression that computes a table. Greenplum Database allows this type only as an argument to a user-defined function. See *Table Value Expressions* for more about the *anytable* pseudo-type.

For more information about pseudo-types, see the PostgreSQL documentation about *Pseudo-Types*.

Polymorphic Types

Four pseudo-types of special interest are *anyelement*, *anyarray*, *anynonarray*, and *anyenum*, which are collectively called *polymorphic* types. Any function declared using these types is said to be a polymorphic function. A polymorphic function can operate on many different data types, with the specific data types being determined by the data types actually passed to it at runtime.

Polymorphic arguments and results are tied to each other and are resolved to a specific data type when a query calling a polymorphic function is parsed. Each position (either argument or return value) declared as *anyelement* is allowed to have any specific actual data type, but in any given call they must all be the same actual type. Each position declared as *anyarray* can have any array data type, but similarly they must all be the same type. If there are positions declared *anyarray* and others declared *anyelement*, the actual array type in the *anyarray* positions must be an array whose elements are the same type appearing in the *anyelement* positions. *anynonarray* is treated exactly the same as *anyelement*, but adds

the additional constraint that the actual type must not be an array type. *anyenum* is treated exactly the same as *anyelement*, but adds the additional constraint that the actual type must be an `enum` type.

When more than one argument position is declared with a polymorphic type, the net effect is that only certain combinations of actual argument types are allowed. For example, a function declared as `equal(anyelement, anyelement)` takes any two input values, so long as they are of the same data type.

When the return value of a function is declared as a polymorphic type, there must be at least one argument position that is also polymorphic, and the actual data type supplied as the argument determines the actual result type for that call. For example, if there were not already an array subscripting mechanism, one could define a function that implements subscripting as `subscript(anyarray, integer) returns anyelement`. This declaration constrains the actual first argument to be an array type, and allows the parser to infer the correct result type from the actual first argument's type. Another example is that a function declared as `myfunc(anyarray) returns anyenum` will only accept arrays of `enum` types.

Note that *anynonarray* and *anyenum* do not represent separate type variables; they are the same type as *anyelement*, just with an additional constraint. For example, declaring a function as `myfunc(anyelement, anyenum)` is equivalent to declaring it as `myfunc(anyenum, anyenum)`: both actual arguments must be the same `enum` type.

A variadic function (one taking a variable number of arguments) is polymorphic when its last parameter is declared as `VARIADIC anyarray`. For purposes of argument matching and determining the actual result type, such a function behaves the same as if you had declared the appropriate number of *anynonarray* parameters.

For more information about polymorphic types, see the PostgreSQL documentation about *Polymorphic Arguments and Return Types*.

Table Value Expressions

The *anytable* pseudo-type declares a function argument that is a table value expression. The notation for a table value expression is a `SELECT` statement enclosed in a `TABLE()` function. You can specify a distribution policy for the table by adding `SCATTER RANDOMLY`, or a `SCATTER BY` clause with a column list to specify the distribution key.

The `SELECT` statement is executed when the function is called and the result rows are distributed to segments so that each segment executes the function with a subset of the result table.

For example, this table expression selects three columns from a table named `customer` and sets the distribution key to the first column:

```
TABLE(SELECT cust_key, name, address FROM customer SCATTER BY 1)
```

The `SELECT` statement may include joins on multiple base tables, `WHERE` clauses, aggregates, and any other valid query syntax.

The *anytable* type is only permitted in functions implemented in the C or C++ languages. The body of the function can access the table using the Greenplum Database Server Programming Interface (SPI) or the Greenplum Partner Connector (GPPC) API.

The *anytable* type is used in some user-defined functions in the Pivotal GPText API. The following GPText example uses the `TABLE` function with the `SCATTER BY` clause in the GPText function `gptext.index()` to populate the index `mydb.mytest.articles` with data from the messages table:

```
SELECT * FROM gptext.index(TABLE(SELECT * FROM mytest.messages
          SCATTER BY distrib_id), 'mydb.mytest.messages');
```

For information about the function `gptext.index()`, see the Pivotal GPText documentation.

Text Search Data Types

Greenplum Database provides two data types that are designed to support full text search, which is the activity of searching through a collection of natural-language *documents* to locate those that best match a *query*. The `tsvector` type represents a document in a form optimized for text search; the `tsquery` type similarly represents a text query. *Using Full Text Search* provides a detailed explanation of this facility, and *Text Search Functions and Operators* summarizes the related functions and operators.

The `tsvector` and `tsquery` types cannot be part of the distribution key of a Greenplum Database table.

tsvector

A `tsvector` value is a sorted list of distinct *lexemes*, which are words that have been *normalized* to merge different variants of the same word (see *Using Full Text Search* for details). Sorting and duplicate-elimination are done automatically during input, as shown in this example:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

To represent lexemes containing whitespace or punctuation, surround them with quotes:

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

(We use dollar-quoted string literals in this example and the next one to avoid the confusion of having to double quote marks within the literals.) Embedded quotes and backslashes must be doubled:

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                    tsvector
-------------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

Optionally, integer *positions* can be attached to lexemes:

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11
 rat:12'::tsvector;
                                  tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12
 'sat':4
```

A position normally indicates the source word's location in the document. Positional information can be used for *proximity ranking*. Position values can range from 1 to 16383; larger numbers are silently set to 16383. Duplicate positions for the same lexeme are discarded.

Lexemes that have positions can further be labeled with a *weight*, which can be `A`, `B`, `C`, or `D`. `D` is the default and hence is not shown on output:

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

Weights are typically used to reflect document structure, for example by marking title words differently from body words. Text search ranking functions can assign different priorities to the different weight markers.

It is important to understand that the `tsvector` type itself does not perform any normalization; it assumes the words it is given are normalized appropriately for the application. For example,

```
select 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
```

For most English-text-searching applications the above words would be considered non-normalized, but tsvector doesn't care. Raw document text should usually be passed through `to_tsvector` to normalize the words appropriately for searching:

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

tsquery

A `tsquery` value stores lexemes that are to be searched for, and combines them honoring the Boolean operators & (AND), | (OR), and ! (NOT). Parentheses can be used to enforce grouping of the operators:

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

In the absence of parentheses, ! (NOT) binds most tightly, and & (AND) binds more tightly than | (OR).

Optionally, lexemes in a `tsquery` can be labeled with one or more weight letters, which restricts them to match only `tsvector` lexemes with matching weights:

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

Also, lexemes in a `tsquery` can be labeled with * to specify prefix matching:

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

This query will match any word in a `tsvector` that begins with "super". Note that prefixes are first processed by text search configurations, which means this comparison returns true:

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
----------
```

```
 t
(1 row)
```

because `postgres` gets stemmed to `postgr`:

```
SELECT to_tsquery('postgres:*');
 to_tsquery
------------
 'postgr':*
(1 row)
```

which then matches `postgraduate`.

Quoting rules for lexemes are the same as described previously for lexemes in `tsvector`; and, as with `tsvector`, any required normalization of words must be done before converting to the `tsquery` type. The `to_tsquery` function is convenient for performing such normalization:

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

Range Types

Range types are data types representing a range of values of some element type (called the range's *subtype*). For instance, ranges of `timestamp` might be used to represent the ranges of time that a meeting room is reserved. In this case the data type is `tsrange` (short for “timestamp range”), and `timestamp` is the subtype. The subtype must have a total order so that it is well-defined whether element values are within, before, or after a range of values.

Range types are useful because they represent many element values in a single range value, and because concepts such as overlapping ranges can be expressed clearly. The use of time and date ranges for scheduling purposes is the clearest example; but price ranges, measurement ranges from an instrument, and so forth can also be useful.

Built-in Range Types

Greenplum Database comes with the following built-in range types:

- `int4range` -- Range of `integer`
- `int8range` -- Range of `bigint`
- `numrange` -- Range of `numeric`
- `tsrange` -- Range of `timestamp without time zone`
- `tstzrange` -- Range of `timestamp with time zone`
- `daterange` -- Range of `date`

In addition, you can define your own range types; see *CREATE TYPE* for more information.

Examples

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- Containment
SELECT int4range(10, 20) @> 3;

-- Overlaps
```

```
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Extract the upper bound
SELECT upper(int8range(15, 25));

-- Compute the intersection
SELECT int4range(10, 20) * int4range(15, 25);

-- Is the range empty?
SELECT isempty(numrange(1, 5));
```

See *Range Functions and Operators* for complete lists of operators and functions on range types.

Inclusive and Exclusive Bounds

Every non-empty range has two bounds, the lower bound and the upper bound. All points between these values are included in the range. An inclusive bound means that the boundary point itself is included in the range as well, while an exclusive bound means that the boundary point is not included in the range.

In the text form of a range, an inclusive lower bound is represented by `[` while an exclusive lower bound is represented by `(` . Likewise, an inclusive upper bound is represented by `]` , while an exclusive upper bound is represented by `)` . (See *Range Functions and Operators* for more details.)

The functions `lower_inc` and `upper_inc` test the inclusivity of the lower and upper bounds of a range value, respectively.

Infinite (Unbounded) Ranges

The lower bound of a range can be omitted, meaning that all points less than the upper bound are included in the range. Likewise, if the upper bound of the range is omitted, then all points greater than the lower bound are included in the range. If both lower and upper bounds are omitted, all values of the element type are considered to be in the range.

This is equivalent to considering that the lower bound is “minus infinity”, or the upper bound is “plus infinity”, respectively. But note that these infinite values are never values of the range's element type, and can never be part of the range. (So there is no such thing as an inclusive infinite bound -- if you try to write one, it will automatically be converted to an exclusive bound.)

Also, some element types have a notion of “infinity”, but that is just another value so far as the range type mechanisms are concerned. For example, in timestamp ranges, `[today,]` means the same thing as `[today,)`. But `[today,infinity]` means something different from `[today,infinity)` -- the latter excludes the special `timestamp` value `infinity`.

The functions `lower_inf` and `upper_inf` test for infinite lower and upper bounds of a range, respectively.

Range Input/Output

The input for a range value must follow one of the following patterns:

```
(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

The parentheses or brackets indicate whether the lower and upper bounds are exclusive or inclusive, as described previously. Notice that the final pattern is `empty`, which represents an empty range (a range that contains no points).

The *lower-bound* may be either a string that is valid input for the subtype, or empty to indicate no lower bound. Likewise, *upper-bound* may be either a string that is valid input for the subtype, or empty to indicate no upper bound.

Each bound value can be quoted using " (double quote) characters. This is necessary if the bound value contains parentheses, brackets, commas, double quotes, or backslashes, since these characters would otherwise be taken as part of the range syntax. To put a double quote or backslash in a quoted bound value, precede it with a backslash. (Also, a pair of double quotes within a double-quoted bound value is taken to represent a double quote character, analogously to the rules for single quotes in SQL literal strings.) Alternatively, you can avoid quoting and use backslash-escaping to protect all data characters that would otherwise be taken as range syntax. Also, to write a bound value that is an empty string, write "", since writing nothing means an infinite bound.

Whitespace is allowed before and after the range value, but any whitespace between the parentheses or brackets is taken as part of the lower or upper bound value. (Depending on the element type, it might or might not be significant.)

Examples:

```
-- includes 3, does not include 7, and does include all points in between
SELECT '[3,7)'::int4range;

-- does not include either 3 or 7, but includes all points in between
SELECT '(3,7)'::int4range;

-- includes only the single point 4
SELECT '[4,4]'::int4range;

-- includes no points (and will be normalized to 'empty')
SELECT '[4,4)'::int4range;
```

Constructing Ranges

Each range type has a constructor function with the same name as the range type. Using the constructor function is frequently more convenient than writing a range literal constant, since it avoids the need for extra quoting of the bound values. The constructor function accepts two or three arguments. The two-argument form constructs a range in standard form (lower bound inclusive, upper bound exclusive), while the three-argument form constructs a range with bounds of the form specified by the third argument. The third argument must be one of the strings `()` , `(]` , `[)` , or `[]` . For example:

```
-- The full form is: lower bound, upper bound, and text argument indicating
-- inclusivity/exclusivity of bounds.
SELECT numrange(1.0, 14.0, '(]');

-- If the third argument is omitted, '[)' is assumed.
SELECT numrange(1.0, 14.0);

-- Although '(]' is specified here, on display the value will be converted
 to
-- canonical form, since int8range is a discrete range type (see below).
SELECT int8range(1, 14, '(]');

-- Using NULL for either bound causes the range to be unbounded on that
 side.
SELECT numrange(NULL, 2.2);
```

Discrete Range Types

A discrete range is one whose element type has a well-defined “step”, such as `integer` or `date`. In these types two elements can be said to be adjacent, when there are no valid values between them. This contrasts with continuous ranges, where it's always (or almost always) possible to identify other element values between two given values. For example, a range over the `numeric` type is continuous, as is a range over `timestamp`. (Even though `timestamp` has limited precision, and so could theoretically be treated as discrete, it's better to consider it continuous since the step size is normally not of interest.)

Another way to think about a discrete range type is that there is a clear idea of a “next” or “previous” value for each element value. Knowing that, it is possible to convert between inclusive and exclusive representations of a range's bounds, by choosing the next or previous element value instead of the one originally given. For example, in an integer range type `[4,8]` and `(3,9)` denote the same set of values; but this would not be so for a range over numeric.

A discrete range type should have a *canonicalization* function that is aware of the desired step size for the element type. The canonicalization function is charged with converting equivalent values of the range type to have identical representations, in particular consistently inclusive or exclusive bounds. If a canonicalization function is not specified, then ranges with different formatting will always be treated as unequal, even though they might represent the same set of values in reality.

The built-in range types `int4range`, `int8range`, and `daterange` all use a canonical form that includes the lower bound and excludes the upper bound; that is, `[)`. User-defined range types can use other conventions, however.

Defining New Range Types

Users can define their own range types. The most common reason to do this is to use ranges over subtypes not provided among the built-in range types. For example, to define a new range type of subtype `float8`:

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

Because `float8` has no meaningful “step”, we do not define a canonicalization function in this example.

Defining your own range type also allows you to specify a different subtype B-tree operator class or collation to use, so as to change the sort ordering that determines which values fall into a given range.

If the subtype is considered to have discrete rather than continuous values, the `CREATE TYPE` command should specify a `canonical` function. The canonicalization function takes an input range value, and must return an equivalent range value that may have different bounds and formatting. The canonical output for two ranges that represent the same set of values, for example the integer ranges `[1, 7]` and `[1, 8)`, must be identical. It doesn't matter which representation you choose to be the canonical one, so long as two equivalent values with different formattings are always mapped to the same value with the same formatting. In addition to adjusting the inclusive/exclusive bounds format, a canonicalization function might round off boundary values, in case the desired step size is larger than what the subtype is capable of storing. For instance, a range type over `timestamp` could be defined to have a step size of an hour, in which case the canonicalization function would need to round off bounds that weren't a multiple of an hour, or perhaps throw an error instead.

In addition, any range type that is meant to be used with GiST or SP-GiST indexes should define a subtype difference, or `subtype_diff`, function. (The index will still work without `subtype_diff`, but it is likely to be considerably less efficient than if a difference function is provided.) The subtype difference

function takes two input values of the subtype, and returns their difference (i.e., *X* minus *Y*) represented as a `float8` value. In our example above, the function `float8mi` that underlies the regular `float8` minus operator can be used; but for any other subtype, some type conversion would be necessary. Some creative thought about how to represent differences as numbers might be needed, too. To the greatest extent possible, the `subtype_diff` function should agree with the sort ordering implied by the selected operator class and collation; that is, its result should be positive whenever its first argument is greater than its second according to the sort ordering.

A less-oversimplified example of a `subtype_diff` function is:

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);

SELECT '[11:10, 23:00]'::timerange;
```

See *CREATE TYPE* for more information about creating range types.

Indexing

GiST and SP-GiST indexes can be created for table columns of range types. For instance, to create a GiST index:

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

A GiST or SP-GiST index can accelerate queries involving these range operators: =, &&, <@, @>, <<, >>, -|-, &<, and &> (see *Range Functions and Operators* for more information).

In addition, B-tree and hash indexes can be created for table columns of range types. For these index types, basically the only useful range operation is equality. There is a B-tree sort ordering defined for range values, with corresponding < and > operators, but the ordering is rather arbitrary and not usually useful in the real world. Range types' B-tree and hash support is primarily meant to allow sorting and hashing internally in queries, rather than creation of actual indexes.

Summary of Built-in Functions

Greenplum Database supports built-in functions and operators including analytic functions and window functions that can be used in window expressions. For information about using built-in Greenplum Database functions see, "Using Functions and Operators" in the *Greenplum Database Administrator Guide*.

Greenplum Database Function Types

Greenplum Database evaluates functions and operators used in SQL expressions. Some functions and operators are only allowed to execute on the master since they could lead to inconsistencies in Greenplum Database segment instances. This table describes the Greenplum Database Function Types.

Table 98: Functions in Greenplum Database

Function Type	Greenplum Support	Description	Comments
IMMUTABLE	Yes	Relies only on information directly in its argument list. Given the same argument values, always returns the same result.	
STABLE	Yes, in most cases	Within a single table scan, returns the same result for same argument values, but results change across SQL statements.	Results depend on database lookups or parameter values. `current_timestamp` family of functions is `STABLE`; values do not change within an execution.
VOLATILE	Restricted	Function values can change within a single table scan. For example: `random()`, `timeofday()`.	Any function with side effects is volatile, even if its result is predictable. For example: `setval()`.

In Greenplum Database, data is divided up across segments — each segment is a distinct PostgreSQL database. To prevent inconsistent or unexpected results, do not execute functions classified as `VOLATILE` at the segment level if they contain SQL commands or modify the database in any way. For example, functions such as `setval()` are not allowed to execute on distributed data in Greenplum Database because they can cause inconsistent data between segment instances.

To ensure data consistency, you can safely use `VOLATILE` and `STABLE` functions in statements that are evaluated on and run from the master. For example, the following statements run on the master (statements without a `FROM` clause):

```
SELECT setval('myseq', 201);
SELECT foo();
```

If a statement has a `FROM` clause containing a distributed table *and* the function in the `FROM` clause returns a set of rows, the statement can run on the segments:

```
SELECT * from foo();
```

Greenplum Database does not support functions that return a table reference (`rangeFuncs`) or functions that use the `refCursor` datatype.

Built-in Functions and Operators

The following table lists the categories of built-in functions and operators supported by PostgreSQL. All functions and operators are supported in Greenplum Database as in PostgreSQL with the exception of `STABLE` and `VOLATILE` functions, which are subject to the restrictions noted in *Greenplum Database Function Types*. See the *Functions and Operators* section of the PostgreSQL documentation for more information about these built-in functions and operators.

Table 99: Built-in functions and operators

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
Logical Operators			
Comparison Operators			
Mathematical Functions and Operators	random setseed		
String Functions and Operators	*All built-in conversion functions*	convert pg_client_encoding	
Binary String Functions and Operators			
Bit String Functions and Operators			
Pattern Matching			
Data Type Formatting Functions		to_char to_timestamp	
Date/Time Functions and Operators	timeofday	age current_date current_time current_timestamp localtime localtimestamp now	
Enum Support Functions			
Geometric Functions and Operators			

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
Network Address Functions and Operators			
Sequence Manipulation Functions	nextval() setval()		
Conditional Expressions			
Array Functions and Operators		*All array functions*	
Aggregate Functions			
Subquery Expressions			
Row and Array Comparisons			
Set Returning Functions	generate_series		
System Information Functions		*All session information functions* *All access privilege inquiry functions* *All schema visibility inquiry functions* *All system catalog information functions* *All comment information functions* *All transaction ids and snapshots*	
System Administration Functions	set_config pg_cancel_backend pg_reload_conf pg_rotate_logfile pg_start_backup pg_stop_backup pg_size_pretty pg_ls_dir pg_read_file pg_stat_file	current_setting *All database object size functions*	**Note:** The function `pg_column_size` displays bytes required to store the value, possibly with TOAST compression.

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
XML Functions and function-like expressions		cursor_to_xml(cursor refcursor, count int, nulls boolean, tableforest boolean, targetns text) cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text) database_to_xml(nulls boolean, tableforest boolean, targetns text) database_to_xmlschema(nulls boolean, tableforest boolean, targetns text) database_to_xml_and_ xmlschema(nulls boolean, tableforest boolean, targetns text) query_to_xml(query text, nulls boolean, tableforest boolean, targetns text) query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text) query_to_xml_and_ xmlschema(query text, nulls boolean, tableforest boolean, targetns text) schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text) schema_to_ xmlschema(schema name, nulls boolean, tableforest boolean, targetns text) schema_to_xml_and_ xmlschema(schema name, nulls boolean, tableforest boolean, targetns text) table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text) table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)	

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
		table_to_xml_and_ xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text) xmlagg(xml) xmlconcat(xml[, ...]) xmlelement(name name [, xmlattributes(value [AS attname] [, ...])] [, content, ...]) xmlexists(text, xml) xmlforest(content [AS name] [, ...]) xml_is_well_formed(text) xml_is_well_formed_ document(text) xml_is_well_formed_ content(text) xmlparse ({ DOCUMENT | CONTENT } value) xpath(text, xml) xpath(text, xml, text[]) xpath_exists(text, xml) xpath_exists(text, xml, text[]) xmlpi(name target [, content]) xmlroot(xml, version text | no value [, standalone yes|no|no value]) xmlserialize ({ DOCUMENT | CONTENT } value AS type) xml(text) text(xml) xmlcomment(xml) xmlconcat2(xml, xml)	

JSON Functions and Operators

Greenplum Database includes built-in functions and operators that create and manipulate JSON data.

- *JSON Operators*
- *JSON Creation Functions*
- *JSON Aggregate Functions*
- *JSON Processing Functions*

Note: For `json` data type values, all key/value pairs are kept even if a JSON object contains duplicate keys. For duplicate keys, JSON processing functions consider the last value as the operative one. For the `jsonb` data type, duplicate object keys are not kept. If the input includes duplicate keys, only the last value is kept. See *About JSON Data* in the *Greenplum Database Administrator Guide*.

JSON Operators

This table describes the operators that are available for use with the `json` and `jsonb` data types.

Table 100: *json* and `jsonb` Operators

Operator	Right Operand Type	Description	Example	Example Result
`->`	`int`	Get the JSON array element (indexed from zero).	`'[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->2`	`{"c":"baz"}`
`->`	`text`	Get the JSON object field by key.	`'{"a": {"b":"foo"}}'::json->'a'`	`{"b":"foo"}`
`->>`	`int`	Get the JSON array element as `text`.	`'[1,2,3]'::json->>2`	`3`
`->>`	`text`	Get the JSON object field as `text`.	`'{"a":1,"b":2}'::json->>'b'`	`2`
`#>`	`text[]`	Get the JSON object at specified path.	`'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}'`	`{"c": "foo"}`
`#>>`	`text[]`	Get the JSON object at specified path as `text`.	`'{"a":[1,2,3],"b": [4,5,6]}'::json#>>'{a,2}'`	`3`

Note: There are parallel variants of these operators for both the `json` and `jsonb` data types. The field, element, and path extraction operators return the same data type as their left-hand input (either `json` or `jsonb`), except for those specified as returning `text`, which coerce the value to `text`. The field, element, and path extraction operators return `NULL`, rather than failing, if the JSON input does not have the right structure to match the request; for example if no such element exists.

Operators that require the `jsonb` data type as the left operand are described in the following table. Many of these operators can be indexed by `jsonb` operator classes. For a full description of `jsonb` containment and existence semantics, see *jsonb Containment and Existence* in the *Greenplum Database Administrator Guide*. For information about how these operators can be used to effectively index `jsonb` data, see *jsonb Indexing* in the *Greenplum Database Administrator Guide*.

Table 101: `jsonb` Operators

Operator	Right Operand Type	Description	Example
`@>`	`jsonb`	Does the left JSON value contain within it the right value?	`'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb`

Operator	Right Operand Type	Description	Example
<@	jsonb	Is the left JSON value contained within the right value?	'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb
?	text	Does the key/element string exist within the JSON value?	'{"a":1, "b":2}'::jsonb ? 'b'
?|	text[]	Do any of these key/element strings exist?	'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']
?&	text[]	Do all of these key/element strings exist?	'["a", "b"]'::jsonb ?& array['a', 'b']

The standard comparison operators in the following table are available only for the jsonb data type, not for the json data type. They follow the ordering rules for B-tree operations described in *jsonb Indexing* in the *Greenplum Database Administrator Guide*.

Table 102: jsonb Comparison Operators

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
<> or !=	not equal

Note: The != operator is converted to <> in the parser stage. It is not possible to implement != and <> operators that do different things.

JSON Creation Functions

This table describes the functions that create json data type values. (Currently, there are no equivalent functions for jsonb, but you can cast the result of one of these functions to jsonb.)

Table 103: JSON Creation Functions

Function	Description	Example	Example Result
`to_json(anyelement)`	Returns the value as a JSON object. Arrays and composites are processed recursively and are converted to arrays and objects. If the input contains a cast from the type to `json`, the cast function is used to perform the conversion; otherwise, a JSON scalar value is produced. For any scalar type other than a number, a Boolean, or a null value, the text representation will be used, properly quoted and escaped so that it is a valid JSON string.	`to_json('Fred said "Hi."'::text)`	`"Fred said \"Hi.\""`
`array_to_json(anyarray [, pretty_bool])`	Returns the array as a JSON array. A multidimensional array becomes a JSON array of arrays. Line feeds will be added between dimension-1 elements if `pretty_bool` is true.	`array_to_json('{{1,5},{99,100}}'::int[])`	`[[1,5],[99,100]]`
`row_to_json(record [, pretty_bool])`	Returns the row as a JSON object. Line feeds will be added between level-1 elements if `pretty_bool` is true.	`row_to_json(row(1,'foo'))`	`{"f1":1,"f2":"foo"}`
`json_build_array(VARIADIC "any")`	Builds a possibly-heterogeneously-typed JSON array out of a `VARIADIC` argument list.	`json_build_array(1,2,'3',4,5)`	`[1, 2, "3", 4, 5]`
`json_build_object(VARIADIC "any")`	Builds a JSON object out of a `VARIADIC` argument list. The argument list is taken in order and converted to a set of key/value pairs.	`json_build_object('foo',1,'bar',2)`	`{"foo": 1, "bar": 2}`

Function	Description	Example	Example Result
`json_object(text[])`	Builds a JSON object out of a text array. The array must be either a one or a two dimensional array. The one dimensional array must have an even number of elements. The elements are taken as key/value pairs. For a two dimensional array, each inner array must have exactly two elements, which are taken as a key/value pair.	`json_object('{a, 1, b, "def", c, 3.5}')` `json_object('{{a, 1},{b, "def"},{c, 3.5}}')`	`{"a": "1", "b": "def", "c": "3.5"}`
`json_object(keys text[], values text[])`	Builds a JSON object out of a text array. This form of `json_object` takes keys and values pairwise from two separate arrays. In all other respects it is identical to the one-argument form.	`json_object('{a, b}', '{1,2}')`	`{"a": "1", "b": "2"}`

Note: `array_to_json` and `row_to_json` have the same behavior as `to_json` except for offering a pretty-printing option. The behavior described for `to_json` likewise applies to each individual value converted by the other JSON creation functions.

Note: The *hstore* extension has a cast from `hstore` to `json`, so that `hstore` values converted via the JSON creation functions will be represented as JSON objects, not as primitive string values.

JSON Aggregate Functions

This table shows the functions aggregate records to an array of JSON objects and pairs of values to a JSON object

Table 104: JSON Aggregate Functions

Function	Argument Types	Return Type	Description
`json_agg(record)`	`record`	`json`	Aggregates records as a JSON array of objects.
`json_object_agg(name, value)`	`("any", "any")`	`json`	Aggregates name/value pairs as a JSON object.

JSON Processing Functions

This table shows the functions that are available for processing `json` and `jsonb` values.

Many of these processing functions and operators convert Unicode escapes in JSON strings to the appropriate single character. This is a not an issue if the input data type is `jsonb`, because the conversion

was already done. However, for `json` data type input, this might result in an error being thrown. See *About JSON Data* in the *Greenplum Database Administrator Guide*.

Table 105: JSON Processing Functions

Function	Return Type	Description	Example	Example Result
`json_array_length(json)` `jsonb_array_length(jsonb)`	`int`	Returns the number of elements in the outermost JSON array.	`json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]')`	`5`

`json_each(json)`
`jsonb_each(jsonb)`
	`setof key text, value json`
	`setof key text, value jsonb`
		Expands the outermost JSON object into a set of key/value pairs.	`select * from json_each('{"a":"foo", "b":"bar"}')`	`key | value`
				`-----+-------`
				`a | "foo"`
				`b | "bar"`

`json_each_text(json)`
`jsonb_each_text(jsonb)`
	`setof key text, value text`	Expands the outermost JSON object into a set of key/value pairs. The returned values will be of type `text`.	`select * from json_each_text('{"a":"foo", "b":"bar"}')`	`key | value`
				`-----+-------`
				`a | foo`
				`b | bar`

`json_extract_path(from_json json, VARIADIC path_elems text[])` `jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])`	`json` `jsonb`	Returns the JSON value pointed to by `path_elems` (equivalent to #> operator).	`json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4')`	`{"f5":99,"f6":"foo"}`
`json_extract_path_text(from_json json, VARIADIC path_elems text[])` `jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])`	`text`	Returns the JSON value pointed to by `path_elems` as text. Equivalent to #>> operator.	`json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4', 'f6')`	`foo`

Function	Return Type	Description	Example	Example Result

`json_object_keys(json)`
`jsonb_object_keys(jsonb)`
	`setof text`	Returns set of keys in the outermost JSON object.	`json_object_keys('{"f1":"abc","f2":{"f3":"a","f4":"b"}}')`	`json_object_keys`
				`----------`
				`f1`
				`f2`

`json_populate_record(base anyelement, from_json json)`
`jsonb_populate_record(base anyelement, from_json jsonb)`
	`anyelement`	Expands the object in `from_json` to a row whose columns match the record type defined by base. See *Note 1*.	`select * from json_populate_record(null::myrow, '{"a":1,"b":2}')`	`a | b`
				`---+---`
				`1 | 2`

`json_populate_recordset(base anyelement, from_json json)`
`jsonb_populate_recordset(base anyelement, from_json jsonb)`
	`setof anyelement`	Expands the outermost array of objects in `from_json` to a set of rows whose columns match the record type defined by base. See *Note 1*.	`select * from json_populate_recordset(null::myrow, '[{"a":1,"b":2},{"a":3,"b":4}]')`	`a | b`
				`---+---`
				`1 | 2`
				`3 | 4`

`json_array_elements(json)`
`jsonb_array_elements(jsonb)`
	`setof json`
	`setof jsonb`
		Expands a JSON array to a set of JSON values.	`select * from json_array_elements('[1,true,[2,false]]')`	`value`
				`-----------`
				`1`
				`true`
				`[2,false]`

`json_array_elements_text(json)`
`jsonb_array_elements_text(jsonb)`
	`setof text`	Expands a JSON array to a set of `text` values.	`select * from json_array_elements_text('["foo","bar"]')`	`value`
				`-----------`
				`foo`
				`bar`

`json_typeof(json)` `jsonb_typeof(jsonb)`	`text`	Returns the type of the outermost JSON value as a text string. Possible types are `object`, `array`, `string`, `number`, `boolean`, and `null`. See *Note 2*	`json_typeof('-123.4')`	`number`

Function	Return Type	Description	Example	Example Result

`json_to_record(json)`
`jsonb_to_record(jsonb)`
	`record`	Builds an arbitrary record from a JSON object. See *Note 1*.
		As with all functions returning record, the caller must explicitly define the structure of the record with an AS clause.
			`select * from json_to_record('{"a":1,"b":[1,2,3],"c":"bar"}') as x(a int, b text, d text)`	`a | b | d`
				`---+---------+---`
				`1 | [1,2,3] |`

`json_to_recordset(json)`
`jsonb_to_recordset(jsonb)`
	`setof record`	Builds an arbitrary set of records from a JSON array of objects See *Note 1*.
		As with all functions returning record, the caller must explicitly define the structure of the record with an AS clause.
			`select * from json_to_recordset('[{"a":1,"b":"foo"},{"a":"2","c":"bar"}]') as x(a int, b text);`	`a | b`
				`---+-----`
				`1 | foo`
				`2 |`

Note:

1. The examples for the functions `json_populate_record()`, `json_populate_recordset()`, `json_to_record()` and `json_to_recordset()` use constants. However, the typical use would be to reference a table in the `FROM` clause and use one of its `json` or `jsonb` columns as an argument to the function. The extracted key values can then be referenced in other parts of the query. For example the value can be referenced in `WHERE` clauses and target lists. Extracting multiple values in this way can improve performance over extracting them separately with per-key operators.

   JSON keys are matched to identical column names in the target row type. JSON type coercion for these functions might not result in desired values for some types. JSON fields that do not appear in the target row type will be omitted from the output, and target columns that do not match any JSON field will be `NULL`.
2. The `json_typeof` function null return value of `null` should not be confused with a SQL `NULL`. While calling `json_typeof('null'::json)` will return `null`, calling `json_typeof(NULL::json)` will return a SQL `NULL`.

Window Functions

The following are Greenplum Database built-in window functions. All window functions are *immutable*. For more information about window functions, see "Window Expressions" in the *Greenplum Database Administrator Guide*.

Table 106: Window functions

Function	Return Type	Full Syntax	Description
`cume_dist()`	`double precision`	`CUME_DIST() OVER ( [PARTITION BY expr ] ORDER BY expr )`	Calculates the cumulative distribution of a value in a group of values. Rows with equal values always evaluate to the same cumulative distribution value.
`dense_rank()`	`bigint`	`DENSE_RANK () OVER ( [PARTITION BY expr ] ORDER BY expr )`	Computes the rank of a row in an ordered group of rows without skipping rank values. Rows with equal values are given the same rank value.
`first_value(`*`expr`*`)`	same as input expr type	`FIRST_VALUE(` expr `) OVER ( [PARTITION BY expr ] ORDER BY expr [ROWS|RANGE frame_expr ] )`	Returns the first value in an ordered set of values.
`lag(`*`expr`* `[,`*`offset`*`] [,`*`default`*`])`	same as input *expr* type	`LAG(` *expr* `[,` *offset* `] [,` *default* `]) OVER ( [PARTITION BY` *expr* `] ORDER BY` *expr* `)`	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, `LAG` provides access to a row at a given physical offset prior to that position. The default `offset` is 1. *default* sets the value that is returned if the offset goes beyond the scope of the window. If *default* is not specified, the default value is null.
`last_value(`*`expr`*`)`	same as input *expr* type	`LAST_VALUE(`*`expr`*`) OVER ( [PARTITION BY` *`expr`*`] ORDER BY` *`expr`* `[ROWS|RANGE` *`frame_expr`*`] )`	Returns the last value in an ordered set of values.

Function	Return Type	Full Syntax	Description
`lead(expr [,offset] [,default])`	same as input *expr* type	`LEAD(expr [,offset] [,exprdefault]) OVER ( [PARTITION BY expr] ORDER BY expr )`	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, `lead` provides access to a row at a given physical offset after that position. If *offset* is not specified, the default offset is 1. *default* sets the value that is returned if the offset goes beyond the scope of the window. If *default* is not specified, the default value is null.
`ntile(expr)`	`bigint`	`NTILE(expr) OVER ( [PARTITION BY expr] ORDER BY expr )`	Divides an ordered data set into a number of buckets (as defined by *expr*) and assigns a bucket number to each row.
`percent_rank()`	`double precision`	`PERCENT_RANK () OVER ( [PARTITION BY expr] ORDER BY expr )`	Calculates the rank of a hypothetical row R minus 1, divided by 1 less than the number of rows being evaluated (within a window partition).
`rank()`	`bigint`	`RANK () OVER ( [PARTITION BY expr] ORDER BY expr )`	Calculates the rank of a row in an ordered group of values. Rows with equal values for the ranking criteria receive the same rank. The number of tied rows are added to the rank number to calculate the next rank value. Ranks may not be consecutive numbers in this case.
`row_number()`	`bigint`	`ROW_NUMBER () OVER ( [PARTITION BY expr] ORDER BY expr )`	Assigns a unique number to each row to which it is applied (either each row in a window partition or each row of the query).

Advanced Aggregate Functions

The following built-in advanced analytic functions are Greenplum extensions of the PostgreSQL database. Analytic functions are *immutable*.

Note: The Greenplum MADlib Extension for Analytics provides additional advanced functions to perform statistical analysis and machine learning with Greenplum Database data. See *MADlib Extension for Analytics*.

Table 107: Advanced Aggregate Functions

Function	Return Type	Full Syntax	Description

`MEDIAN (expr)`	`timestamp`, `timestamptz`, `interval`, `float`	`MEDIAN (expression)`
		Example:
		`SELECT department_id, MEDIAN(salary) FROM employees GROUP BY department_id;`
			Can take a two-dimensional array as input. Treats such arrays as matrices.

`PERCENTILE_CONT (expr) WITHIN GROUP (ORDER BY expr [DESC/ ASC])`	`timestamp`, `timestamptz`, `interval`, `float`	`PERCENTILE_CONT(percentage) WITHIN GROUP (ORDER BY expression)`
		Example:
		`SELECT department_id, PERCENTILE_CONT (0.5) WITHIN GROUP (ORDER BY salary DESC) "Median_cont"; FROM employees GROUP BY department_id;`
			Performs an inverse distribution function that assumes a continuous distribution model. It takes a percentile value and a sort specification and returns the same datatype as the numeric datatype of the argument. This returned value is a computed result after performing linear interpolation. Null are ignored in this calculation.

`PERCENTILE_DISC (expr) WITHIN GROUP (ORDER BY expr [DESC/ ASC])`	`timestamp`, `timestamptz`, `interval`, `float`	`PERCENTILE_DISC(percentage) WITHIN GROUP (ORDER BY expression)`
		Example:
		`SELECT department_id, PERCENTILE_DISC (0.5) WITHIN GROUP (ORDER BY salary DESC) "Median_desc"; FROM employees GROUP BY department_id;`
			Performs an inverse distribution function that assumes a discrete distribution model. It takes a percentile value and a sort specification. This returned value is an element from the set. Null are ignored in this calculation.

Function	Return Type	Full Syntax	Description

`sum(array[])`	`smallint[]`, `bigint[]`, `float[]`	`sum(array[[1,2],[3,4]])`
		Example:
		`CREATE TABLE mymatrix (myvalue int[]);`
		`INSERT INTO mymatrix VALUES (array[[1,2],[3,4]]);`
		`INSERT INTO mymatrix VALUES (array[[0,1],[1,0]]);`
		`SELECT sum(myvalue) FROM mymatrix;`
		`sum`
		`---------------`
		`{{1,3},{4,4}}`
			Performs matrix summation. Can take as input a two-dimensional array that is treated as a matrix.

`pivot_sum (label[], label, expr)`	`int[]`, `bigint[]`, `float[]`	`pivot_sum( array['A1','A2'], attr, value)`	A pivot aggregation using sum to resolve duplicate entries.
`unnest (array[])`	set of `anyelement`	`unnest( array['one', 'row', 'per', 'item'])`	Transforms a one dimensional array into rows. Returns a set of `anyelement`, a polymorphic *pseudotype in PostgreSQL*.

Text Search Functions and Operators

The following tables summarize the functions and operators that are provided for full text searching. See *Using Full Text Search* for a detailed explanation of Greenplum Database's text search facility.

Table 108: Text Search Operators

Operator	Description	Example	Result
`@@`	`tsvector` matches `tsquery` ?	`to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')`	`t`
`@@@`	deprecated synonym for `@@`	`to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat')`	`t`
`||`	concatenate `tsvectors`	`'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector`	`'a':1 'b':2,5 'c':3 'd':4`
`&&`	AND `tsquerys` together	`'fat | rat'::tsquery && 'cat'::tsquery`	`( 'fat' | 'rat' ) & 'cat'`

Operator	Description	Example	Result
`||`	OR `tsquery`s together	`'fat | rat'::tsquery || 'cat'::tsquery`	`( 'fat' | 'rat' ) | 'cat'`
`!!`	negate a `tsquery`	`!! 'cat'::tsquery`	`!'cat'`
`@>`	`tsquery` contains another ?	`'cat'::tsquery @> 'cat & rat'::tsquery`	`f`
`<@`	`tsquery` is contained in ?	`'cat'::tsquery <@ 'cat & rat'::tsquery`	`t`

Note: The `tsquery` containment operators consider only the lexemes listed in the two queries, ignoring the combining operators.

In addition to the operators shown in the table, the ordinary B-tree comparison operators (=, <, etc) are defined for types `tsvector` and `tsquery`. These are not very useful for text searching but allow, for example, unique indexes to be built on columns of these types.

Table 109: Text Search Functions

Function	Return Type	Description	Example	Result
`get_current_ts_config()`	regconfig	get default text search configuration	get_current_ts_config()	english
`length(tsvector)`	integer	number of lexemes in tsvector	length('fat:2,4 cat:3 rat:5A'::tsvector)	3
`numnode(tsquery)`	integer	number of lexemes plus operators in tsquery	numnode('(fat & rat) | cat'::tsquery)	5
`plainto_tsquery([ config regconfig , ] querytext)`	tsquery	produce tsquery ignoring punctuation	plainto_tsquery('english', 'The Fat Rats')	'fat' & 'rat'
`querytree(query tsquery)`	text	get indexable part of a tsquery	querytree('foo & ! bar'::tsquery)	'foo'
`setweight(tsvector, "char")`	tsvector	assign weight to each element of tsvector	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')	'cat':3A 'fat':2A,4A 'rat':5A
`strip(tsvector)`	tsvector	remove positions and weights from tsvector	strip('fat:2,4 cat:3 rat:5A'::tsvector)	'cat' 'fat' 'rat'
`to_tsquery([ config regconfig , ] query text)`	tsquery	normalize words and convert to tsquery	to_tsquery('english', 'The & Fat & Rats')	'fat' & 'rat'

Function	Return Type	Description	Example	Result
to_tsvector([config regconfig ,] documenttext)	tsvector	reduce document text to tsvector	to_tsvector('english', 'The Fat Rats')	'fat':2 'rat':3
ts_headline([config regconfig,] documenttext, query tsquery [, options text])	text	display a query match	ts_headline('x y z', 'z'::tsquery)	x y z
ts_rank([weights float4[],] vector tsvector,query tsquery [, normalization integer])	float4	rank document for query	ts_rank(textsearch, query)	0.818
ts_rank_cd([weights float4[],] vectortsvector, query tsquery [, normalizationinteger])	float4	rank document for query using cover density	ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query)	2.01317
ts_rewrite(query tsquery, target tsquery,substitute tsquery)	tsquery	replace target with substitute within query	ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery)	'b' & ('foo' | 'bar')
ts_rewrite(query tsquery, select text)	tsquery	replace using targets and substitutes from a SELECTcommand	SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')	'b' & ('foo' | 'bar')
tsvector_update_trigger()	trigger	trigger function for automatic tsvector column update	CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)	
tsvector_update_trigger_column()	trigger	trigger function for automatic tsvector column update	CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, configcol, title, body)	

Note: All the text search functions that accept an optional `regconfig` argument will use the configuration specified by *default_text_search_config* when that argument is omitted.

The functions in the following table are listed separately because they are not usually used in everyday text searching operations. They are helpful for development and debugging of new text search configurations.

Table 110: Text Search Debugging Functions

Function	Return Type	Description	Example	Result
ts_debug([*config* regconfig,] *document* text, OUT *alias* text, OUT *description* text, OUT *token* text, OUT *dictionaries* regdictionary[], OUT *dictionary* regdictionary, OUT *lexemes* text[])	setof record	test a configuration	ts_debug('english', 'The Brightest supernovaes')	(asciiword,"Word, all ASCII",The, {english_stem},english_stem,{}) ...
ts_lexize(*dict* regdictionary, *token* text)	text[]	test a dictionary	ts_lexize('english_stem', 'stars')	{star}
ts_parse(*parser_name* text, *document* text, OUT *tokid* integer, OUT *token* text)	setof record	test a parser	ts_parse('default', 'foo - bar')	(1,foo) ...
ts_parse(*parser_oid* oid, *document* text, OUT *tokid* integer, OUT *token* text)	setof record	test a parser	ts_parse(3722, 'foo - bar')	(1,foo) ...
ts_token_type(*parser_name* text, OUT *tokid* integer, OUT *alias* text, OUT description text)	setof record	get token types defined by parser	ts_token_type('default')	(1,asciiword,"Word, all ASCII") ...

Function	Return Type	Description	Example	Result
`ts_token_type(`*`parser_oid`* `oid, OUT` *`tokid`* `integer, OUT` *`alias`* `text, OUT` *`description`* `text)`	`setof record`	get token types defined by parser	`ts_token_type(3722)`	`(1,asciiword,"Word, all ASCII") ...`
`ts_stat(`*`sqlquery`* `text, [` *`weights`* `text, ] OUT` *`word`* `text, OUT` *`ndocinteger`*`, OUT` *`nentry`* `integer)`	`setof record`	get statistics of a tsvectorcolumn	`ts_stat('SELECT vector from apod')`	`(foo,10,15) ...`

Range Functions and Operators

See *Range Types* for an overview of range types.

The following table shows the operators available for range types.

Table 111: Range Operators

Operator	Description	Example	Result
`=`	equal	`int4range(1,5) = '[1,4]'::int4range`	`t`
`<>`	not equal	`numrange(1.1,2.2) <> numrange(1.1,2.3)`	`t`
`<`	less than	`int4range(1,10) < int4range(2,3)`	`t`
`>`	greater than	`int4range(1,10) > int4range(1,5)`	`t`
`<=`	less than or equal	`numrange(1.1,2.2) <= numrange(1.1,2.2)`	`t`
`>=`	greater than or equal	`numrange(1.1,2.2) >= numrange(1.1,2.0)`	`t`
`@>`	contains range	`int4range(2,4) @> int4range(2,3)`	`t`
`@>`	contains element	`'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp`	`t`

Operator	Description	Example	Result
<@	range is contained by	`int4range(2,4) <@ int4range(1,7)`	t
<@	element is contained by	`42 <@ int4range(1,7)`	f
&&	overlap (have points in common)	`int8range(3,7) && int8range(4,12)`	t
<<	strictly left of	`int8range(1,10) << int8range(100,110)`	t
>>	strictly right of	`int8range(50,60) >> int8range(20,30)`	t
&<	does not extend to the right of	`int8range(1,20) &< int8range(18,20)`	t
&>	does not extend to the left of	`int8range(7,20) &> int8range(5,10)`	t
-|-	is adjacent to	`numrange(1.1,2.2) -|- numrange(2.2,3.3)`	t
+	union	`numrange(5,15) + numrange(10,20)`	[5,20)
*	intersection	`int8range(5,15) * int8range(10,20)`	[10,15)
-	difference	`int8range(5,15) - int8range(10,20)`	[5,10)

The simple comparison operators <, >, <=, and >= compare the lower bounds first, and only if those are equal, compare the upper bounds. These comparisons are not usually very useful for ranges, but are provided to allow B-tree indexes to be constructed on ranges.

The left-of/right-of/adjacent operators always return false when an empty range is involved; that is, an empty range is not considered to be either before or after any other range.

The union and difference operators will fail if the resulting range would need to contain two disjoint sub-ranges, as such a range cannot be represented.

The following table shows the functions available for use with range types.

Table 112: Range Functions

Function	Return Type	Description	Example	Result
`lower(anyrange)`	range's element type	lower bound of range	`lower(numrange(1.1,2.2))`	1.1
`upper(anyrange)`	range's element type	upper bound of range	`upper(numrange(1.1,2.2))`	2.2
`isempty(anyrange)`	boolean	is the range empty?	`isempty(numrange(1.1,2.2))`	false

Function	Return Type	Description	Example	Result
`lower_inc(anyrange)`	`boolean`	is the lower bound inclusive?	`lower_inc(numrange(1.1,2.2))`	`true`
`upper_inc(anyrange)`	`boolean`	is the upper bound inclusive?	`upper_inc(numrange(1.1,2.2))`	`false`
`lower_inf(anyrange)`	`boolean`	is the lower bound infinite?	`lower_inf('(,)'::daterange)`	`true`
`upper_inf(anyrange)`	`boolean`	is the upper bound infinite?	`upper_inf('(,)'::daterange)`	`true`
`range_merge(anyrange, anyrange)`	`anyrange`	the smallest range which includes both of the given ranges	`range_merge('[1,2)'::int4range, '[3,4)'::int4range)`	`[1,4)`

The `lower` and `upper` functions return null if the range is empty or the requested bound is infinite. The `lower_inc`, `upper_inc`, `lower_inf`, and `upper_inf` functions all return false for an empty range.

Additional Supplied Modules

This section describes additional modules available in the Greenplum Database installation. These modules may be PostgreSQL- or Greenplum-sourced.

The following Greenplum Database and PostgreSQL `contrib` modules are installed; refer to the linked module documentation for usage instructions.

- *`auto_explain`* Provides a means for logging execution plans of slow statements automatically.
- *`citext`* - Provides a case-insensitive, multibyte-aware text data type.
- *`dblink`* - Provides connections to other Greenplum databases.
- *`diskquota`* - Allows administrators to set disk usage quotas for Greenplum Database roles and schemas.
- *`fuzzystrmatch`* - Determines similarities and differences between strings.
- *`gp_sparse_vector`* - Implements a Greenplum Database data type that uses compressed storage of zeros to make vector computations on floating point numbers faster.
- *`hstore`* - Provides a data type for storing sets of key/value pairs within a single PostgreSQL value.
- *`orafce`* - Provides Greenplum Database-specific Oracle SQL compatibility functions.
- *`pageinspect`* - Provides functions for low level inspection of the contents of database pages; available to superusers only.
- *`pgcrypto`* - Provides cryptographic functions for Greenplum Database.
- *`sslinfo`* - Provides information about the SSL certificate that the current client provided when connecting to Greenplum.

auto_explain

The `auto_explain` module provides a means for logging execution plans of slow statements automatically, without having to run `EXPLAIN` by hand.

The Greenplum Database `auto_explain` module was runs only on the Greenplum Database master segment host. It is otherwise equivalent in functionality to the PostgreSQL `auto_explain` module.

Loading the Module

The `auto_explain` module provides no SQL-accessible functions. To use it, simply load it into the server. You can load it into an individual session by entering this command as a superuser:

```
LOAD 'auto_explain';
```

More typical usage is to preload it into some or all sessions by including `auto_explain` in `session_preload_libraries` or `shared_preload_libraries` in `postgresql.conf`. Then you can track unexpectedly slow queries no matter when they happen. However, this does introduce overhead for all queries.

Module Documentation

See *`auto_explain`* in the PostgreSQL documentation for detailed information about the configuration parameters that control this module's behavior.

citext

The citext module provides a case-insensitive character string data type, `citext`. Essentially, it internally calls the `lower()` function when comparing values. Otherwise, it behaves almost exactly like the `text` data type.

The Greenplum Database `citext` module is equivalent to the PostgreSQL `citext` module. There are no Greenplum Database or MPP-specific considerations for the module.

Installing and Registering the Module

The `citext` module is installed when you install Greenplum Database. Before you can use any of the data types, operators, or functions defined in the module, you must register the `citext` extension in each database in which you want to use the objects. Refer to *Installing Additional Supplied Modules* for more information.

Module Documentation

See *citext* in the PostgreSQL documentation for detailed information about the data types, operators, and functions defined in this module.

dblink

The `dblink` module supports connections to other Greenplum Database databases from within a database session. These databases can reside in the same Greenplum Database system, or in a remote system.

Greenplum Database supports `dblink` connections between databases in Greenplum Database installations with the same major version number. You can also use `dblink` to connect to other Greenplum Database installations that use compatible `libpq` libraries.

> **Note:** `dblink` is intended for database users to perform short ad hoc queries in other databases. `dblink` is not intended for use as a replacement for external tables or for administrative tools such as `gpcopy`.

The Greenplum Database `dblink` module is a modified version of the PostgreSQL `dblink` module. There are some restrictions and limitations when you use the module in Greenplum Database.

Installing and Registering the Module

The `dblink` module is installed when you install Greenplum Database. Before you can use any of the functions defined in the module, you must register the `dblink` extension in each database in which you want to use the functions. Refer to *Installing Additional Supplied Modules* for more information.

Greenplum Database Considerations

In this release of Greenplum Database, statements that modify table data cannot use named or implicit `dblink` connections. Instead, you must provide the connection string directly in the `dblink()` function. For example:

```
gpadmin=# CREATE TABLE testdbllocal (a int, b text) DISTRIBUTED BY (a);
CREATE TABLE
gpadmin=# INSERT INTO testdbllocal select * FROM dblink('dbname=postgres',
 'SELECT * FROM testdblink') AS dbltab(id int, product text);
INSERT 0 2
```

The Greenplum Database version of `dblink` disables the following asynchronous functions:

- `dblink_send_query()`
- `dblink_is_busy()`
- `dblink_get_result()`

Using dblink

The following procedure identifies the basic steps for configuring and using `dblink` in Greenplum Database. The examples use `dblink_connect()` to create a connection to a database and `dblink()` to execute an SQL query.

1. Begin by creating a sample table to query using the `dblink` functions. These commands create a small table in the `postgres` database, which you will later query from the `testdb` database using `dblink`:

```
$ psql -d postgres
psql (9.4.20)
Type "help" for help.

postgres=# CREATE TABLE testdblink (a int, b text) DISTRIBUTED BY (a);
CREATE TABLE
postgres=# INSERT INTO testdblink VALUES (1, 'Cheese'), (2, 'Fish');
INSERT 0 2
postgres=# \q
$
```

2. Log into a different database as a superuser. In this example, the superuser `gpadmin` logs into the database `testdb`. If the `dblink` functions are not already available, register the `dblink` extension in the database:

```
$ psql -d testdb
psql (9.4beta1)
Type "help" for help.

testdb=# CREATE EXTENSION dblink;
CREATE EXTENSION
```

3. Use the `dblink_connect()` function to create either an implicit or a named connection to another database. The connection string that you provide should be a `libpq`-style keyword/value string. This example creates a connection named `mylocalconn` to the `postgres` database on the local Greenplum Database system:

```
testdb=# SELECT dblink_connect('mylocalconn', 'dbname=postgres
 user=gpadmin');
 dblink_connect
----------------
 OK
(1 row)
```

   Note: If a `user` is not specified, `dblink_connect()` uses the value of the `PGUSER` environment variable when Greenplum Database was started. If `PGUSER` is not set, the default is the system user that started Greenplum Database.

4. Use the `dblink()` function to query a database using a configured connection. Keep in mind that this function returns a record type, so you must assign the columns returned in the `dblink()` query. For example, the following command uses the named connection to query the table you created earlier:

```
testdb=# SELECT * FROM dblink('mylocalconn', 'SELECT * FROM testdblink')
 AS dbltab(id int, product text);
 id | product
----+---------
  1 | Cheese
  2 | Fish
(2 rows)
```

To connect to the local database as another user, specify the `user` in the connection string. This example connects to the database as the user `test_user`. Using `dblink_connect()`, a superuser can create a connection to another local database without specifying a password.

```
testdb=# SELECT dblink_connect('localconn2', 'dbname=postgres
 user=test_user');
```

To make a connection to a remote database system, include host and password information in the connection string. For example, to create an implicit `dblink` connection to a remote system:

```
testdb=# SELECT dblink_connect('host=remotehost port=5432 dbname=postgres
 user=gpadmin password=secret');
```

Using dblink as a Non-Superuser

To make a connection to a database with `dblink_connect()`, non-superusers must include host, user, and password information in the connection string. The host, user, and password information must be included even when connecting to a local database. For example, the user `test_user` can create a `dblink` connection to the local system `mdw` with this command:

```
testdb=> SELECT dblink_connect('host=mdw port=5432 dbname=postgres
 user=test_user password=secret');
```

If non-superusers need to create `dblink` connections that do not require a password, they can use the `dblink_connect_u()` function. The `dblink_connect_u()` function is identical to `dblink_connect()`, except that it allows non-superusers to create connections that do not require a password.

`dblink_connect_u()` is initially installed with all privileges revoked from `PUBLIC`, making it un-callable except by superusers. In some situations, it may be appropriate to grant `EXECUTE` permission on `dblink_connect_u()` to specific users who are considered trustworthy, but this should be done with care.

> **Warning:** If a Greenplum Database system has configured users with an authentication method that does not involve a password, then impersonation and subsequent escalation of privileges can occur when a non-superuser executes `dblink_connect_u()`. The `dblink` connection will appear to have originated from the user specified by the function. For example, a non-superuser can execute `dblink_connect_u()` and specify a user that is configured with `trust` authentication.
>
> Also, even if the `dblink` connection requires a password, it is possible for the password to be supplied from the server environment, such as a `~/.pgpass` file belonging to the server's user. It is recommended that any `~/.pgpass` file belonging to the server's user not contain any records specifying a wildcard host name.

1. As a superuser, grant the `EXECUTE` privilege on the `dblink_connect_u()` functions in the user database. This example grants the privilege to the non-superuser `test_user` on the functions with the signatures for creating an implicit or a named `dblink` connection.

   ```
   testdb=# GRANT EXECUTE ON FUNCTION dblink_connect_u(text) TO test_user;
   testdb=# GRANT EXECUTE ON FUNCTION dblink_connect_u(text, text) TO
    test_user;
   ```

2. Now `test_user` can create a connection to another local database without a password. For example, `test_user` can log into the `testdb` database and execute this command to create a connection named `testconn` to the local `postgres` database.

   ```
   testdb=> SELECT dblink_connect_u('testconn', 'dbname=postgres
    user=test_user');
   ```

Note: If a `user` is not specified, `dblink_connect_u()` uses the value of the PGUSER environment variable when Greenplum Database was started. If PGUSER is not set, the default is the system user that started Greenplum Database.

3. `test_user` can use the `dblink()` function to execute a query using a `dblink` connection. For example, this command uses the `dblink` connection named `testconn` created in the previous step. `test_user` must have appropriate access to the table.

```
testdb=> SELECT * FROM dblink('testconn', 'SELECT * FROM testdblink') AS
 dbltab(id int, product text);
```

Using dblink with SSL-Encrypted Connections to Greenplum

When you use `dblink` to connect to Greenplum Database over an encrypted connection, you must specify the `sslmode` property in the connection string. Set `sslmode` to at least `require` to disallow unencrypted transfers. For example:

```
testdb=# SELECT dblink_connect('greenplum_con_sales', 'dbname=sales
 host=gpmaster user=gpadmin sslmode=require');
```

Refer to *SSL Client Authentication* for information about configuring Greenplum Database to use SSL.

Additional Module Documentation

Refer to the *dblink* PostgreSQL documentation for detailed information about the individual functions in this module.

diskquota

The `diskquota` module allows Greenplum Database administrators to limit the amount of disk space used by schemas or roles in a database.

Installing and Registering the Module

The `diskquota` module is installed when you install Greenplum Database.

Before you can use the module, you must perform these steps:

1. Create the `diskquota` database. The `diskquota` module uses this database to store the list of databases where the module is enabled.

```
$ createdb diskquota;
```

2. Add the `diskquota` shared library to the Greenplum Database `shared_preload_libraries` server configuration parameter and restart Greenplum Database. Be sure to retain the previous setting of the configuration parameter. For example:

```
$ gpconfig -s shared_preload_libraries
Values on all segments are consistent
GUC          : shared_preload_libraries
Master  value: auto_explain
Segment value: auto_explain
$ gpconfig -c shared_preload_libraries -v 'auto_explain,diskquota'
$ gpstop -ar
```

3. Register the `diskquota` extension in databases where you want to enforce disk usage quotas. `diskquota` can be registered in up to ten databases.

```
$ psql -d testdb -c "CREATE EXTENSION diskquota"
```

4. If you register the `diskquota` extension in a database that already contains data, you must initialize the `diskquota` table size data by running the `diskquota.init_table_size_table()` UDF in the database. In a database with many files, this can take a long time. The `diskquota` module cannot be used until the initialization is complete.

```
=# SELECT diskquota.init_table_size_table();
```

About the diskquota Module

A Greenplum Database superuser can set disk usage quotas for schemas and roles. A schema quota sets a limit on disk space used by all tables that belong to a schema. A role quota sets a limit on disk space used by all tables that are owned by a role.

Diskquota processes running on the master and segment hosts check disk usage periodically and place schemas or roles on a denylist when they reach their quota.

When a query plan has been generated for a query that would add data, and the schema or role is on the denylist, the query is cancelled before it can start. An error message reports the quota that has been exceeded. A query that does not add data, such as a simple `SELECT` query, is allowed to run even when the role or schema is on the denylist.

Diskquota enforces *soft limits* for disk usage. Quotas are only checked before a query executes. If the quota is not exceeded when the query is about to run, the query is allowed to run, even if it causes the quota to be exceeded.

There is some delay after a quota has been reached before the schema or role is added to the denylist. Other queries could add more data during the delay. The delay occurs because `diskquota` processes that calculate the disk space used by each table execute periodically with a pause between executions (two seconds by default). The delay also occurs when disk usage falls beneath a quota, due to operations such as `DROP`, `TRUNCATE`, or `VACUUM FULL` that remove data. Administrators can change the amount of time between disk space checks by setting the `diskquota.naptime` server configuration parameter.

If a query is unable to run because the schema or role has been denylisted, an administrator can increase the exceeded quota to allow the query to execute. The `show_fast_schema_quota_view` and `show_fast_role_quota_view` views can be used to find the schemas or roles that have exceeded their limits.

Using the diskquota Module

Setting Disk Quotas

Use the `diskquota.set_schema_quota()` and `diskquota.set_role_quota()` user-defined functions in a database to set, update, or delete disk quota limits for schemas and roles in the database. The functions take two arguments: the schema or role name, and the quota to set. The quota can be specified in units of MB, GB, TB, or PB, for example '2TB'.

The following example sets a 250GB quota for the `acct` schema:

```
$ SELECT diskquota.set_schema_quota('acct', '250GB');
```

This example sets a 500MB quota for the `nickd` role:

```
$ SELECT diskquota.set_role_quota(nickd, '500MB');
```

To change a quota, call the `diskquota.set_schema_quota()` or `diskquota.set_role_quota()` function again with the new quota value.

To remove a quota, set the quota value to `'-1'`.

Displaying Disk Quotas and Disk Usage

The `diskquota` module provides two views to display active quotas and the current computed disk space used.

The `diskquota.show_fast_schema_quota_view` view lists active quotas for schemas in the current database. The `nspsize_in_bytes` column contains the calculated size for all tables that belong to the schema.

```
=# SELECT * FROM diskquota.show_fast_schema_quota_view;
 schema_name | schema_oid | quota_in_mb | nspsize_in_bytes
-------------+------------+-------------+------------------
 acct        |      16561 |      256000 |           131072
 analytics   |      16519 |  1073741824 |        144670720
 eng         |      16560 |     5242880 |        117833728
 public      |       2200 |         250 |          3014656
(4 rows)
```

The `diskquota.show_fast_role_quota_view` view lists the active quotas for roles in the current database. The `rolsize_in_bytes` column contains the calculated size for all tables that are owned by the role.

```
=# SELECT * FROM diskquota.show_fast_role_quota_view;
 role_name | role_oid | quota_in_mb | rolsize_in_bytes
-----------+----------+-------------+------------------
 mdach     |    16558 |         500 |           131072
 adam      |    16557 |         300 |        117833728
 nickd     |    16577 |         500 |        144670720
(3 rows)
```

Setting the Delay Between Disk Usage Updates

The `diskquota.naptime` server configuration parameter specifies how frequently (in seconds) the table sizes are recalculated. The smaller the `naptime` value, the less delay in detecting changes in disk usage. This example sets the `naptime` to ten seconds.

```
$ gpconfig -c diskquota.naptime -v 10
```

Configuring diskquota Shared Memory

The `diskquota` module uses shared memory to save the denylist and to save the active table list.

The denylist shared memory can hold up to 1MiB of database objects that exceed the quota limit. If the denylist shared memory fills, data may be loaded into some schemas or roles after they have reached their quota limit.

Active table shared memory holds up to 1MiB of active tables by default. Active tables are tables that may have changed sizes since `diskquota` last recalculated the table sizes. `diskquota` hook functions are called when the storage manager on each Greenplum Database segment creates, extends, or truncates a table file. The hook functions store the identity of the file in shared memory so that its file size can be recalculated the next time the table size data is refreshed.

If the shared memory for active tables fills, `diskquota` may fail to detect a change in disk usage. The amount of active table shared memory can be adjusted by setting the `diskquota.max_active_tables` server configuration parameter. This example changes the active table shared memory to 2MiB:

```
$ gpconfig -c diskquota.max_active_tables -v '2MiB'
```

Shared memory is allocated when Greenplum Database starts up, so a server restart is required after you change the value of the `diskquota.max_active_tables` parameter.

Notes

The `diskquota` module can be enabled in up to ten databases. One diskquota worker process is created on the Greenplum Database master host for each diskquota-enabled database.

The disk usage for a role is defined as the total of disk usage on all segments for all tables the role owns. Although a role is a cluster-level database object, the disk usage for roles is calculated separately for each database.

The disk usage of a schema is defined as the total of disk usage on all segments for all tables in the schema.

The disk usage for a table includes the table data, indexes, toast tables, and free space map. For append-optimized tables, the calculation includes the visibility map and index, and the block directory table.

The `diskquota` module cannot detect a newly created table inside of an uncommited transaction. The size of the new table is not included in the disk usage calculated for the corresponding schema or role until after the transaction has committed. Similarly, a table created using the `CREATE TABLE AS` command is not included in disk usage statistics until the command has completed.

Deleting rows or running `VACUUM` on a table does not release disk space, so these operations cannot alone remove a schema or role from the `diskquota` denylist. The disk space used by a table can be reduced by running `VACUUM FULL` or `TRUNCATE TABLE`.

The `diskquota` module supports high availability features provided by the background worker framework. The `diskquota` launcher process only runs on the active master node. The postmaster on the standby master does not start the `diskquota` launcher process when it is in standby mode. When the master is down and the administrator runs the *`gpactivatestandby`* command, the standby master changes its role to master and the `diskquota` launcher process is forked automatically. Using the `diskquota`-enabled database list in the `diskquota` database, the `diskquota` launcher creates the `diskquota` worker processes that manage disk quotas for each database.

Examples

This example demonstrates how to set up a schema quota and then observe diskquota behavior as data is added to the schema.

1. Create a database named `test` and log in to it.

```
$ createdb test
$ psql -d test
```

2. Create the diskquota extension in the database.

```
=# CREATE EXTENSION diskquota;
CREATE EXTENSION
```

3. Create the `s1` schema.

```
=# CREATE SCHEMA s1;
CREATE SCHEMA
```

4. Set a 1MB disk quota for the `s1` schema.

```
=# SELECT diskquota.set_schema_quota('s1', '1MB');
 set_schema_quota
------------------

(1 row)
```

5. The following commands create a table in the `s1` schema and insert a small amount of data into it. The schema has no data yet, so it is not on the denylist.

```
=# SET search_path TO s1;
SET
=# CREATE TABLE a(i int);
CREATE TABLE
=# INSERT INTO a SELECT generate_series(1,100);
INSERT 0 100
```

6. This command inserts a large amount of data, enough to exceed the 1MB quota that was set for the schema. Before the `INSERT` command, the `s1` schema is still not on the denylist, so this command should be allowed to run, even though it will exceed the limit set for the schema.

```
=# INSERT INTO a SELECT generate_series(1,10000000);
INSERT 0 10000000
```

7. This command attempts to insert a small amount of data. Because the previous command exceeded the schema's disk quota limit, the schema should be denylisted and any data loading command should be cancelled.

```
=# INSERT INTO a SELECT generate_series(1,100);
ERROR:  schema's disk space quota exceeded with name:s1
```

8. This command removes the quota from the `s1` schema by setting it to `-1` and again inserts a small amount of data. A 5-second sleep before the `INSERT` command ensures that the `diskquota` table size data is updated before the command is run.

```
=# SELECT diskquota.set_schema_quota('s1', '-1');
 set_schema_quota
------------------

(1 row)
# Wait for 5 seconds to ensure the denylist is updated
#= SELECT pg_sleep(5);
#= INSERT INTO a SELECT generate_series(1,100);
INSERT 0 100
```

fuzzystrmatch

The `fuzzystrmatch` module provides functions to determine similarities and distance between strings based on various algorithms.

The Greenplum Database `fuzzystrmatch` module is equivalent to the PostgreSQL `fuzzystrmatch` module. There are no Greenplum Database or MPP-specific considerations for the module.

Installing and Registering the Module

The `fuzzystrmatch` module is installed when you install Greenplum Database. Before you can use any of the functions defined in the module, you must register the `fuzzystrmatch` extension in each database in which you want to use the functions. Refer to *Installing Additional Supplied Modules* for more information.

Module Documentation

See *fuzzystrmatch* in the PostgreSQL documentation for detailed information about the individual functions in this module.

gp_sparse_vector

The gp_sparse_vector module implements a Greenplum Database data type and associated functions that use compressed storage of zeros to make vector computations on floating point numbers faster.

The gp_sparse_vector module is a Greenplum Database extension.

Installing and Registering the Module

The gp_sparse_vector module is installed when you install Greenplum Database. Before you can use any of the functions defined in the module, you must register the gp_sparse_vector extension in each database where you want to use the functions. Refer to *Installing Additional Supplied Modules* for more information.

Using the gp_sparse_vector Module

When you use arrays of floating point numbers for various calculations, you will often have long runs of zeros. This is common in scientific, retail optimization, and text processing applications. Each floating point number takes 8 bytes of storage in memory and/or disk. Saving those zeros is often impractical. There are also many computations that benefit from skipping over the zeros.

For example, suppose the following array of doubles is stored as a float8[] in Greenplum Database:

```
'{0, 33, <40,000 zeros>, 12, 22 }'::float8[]
```

This type of array arises often in text processing, where a dictionary may have 40-100K terms and the number of words in a particular document is stored in a vector. This array would occupy slightly more than 320KB of memory/disk, most of it zeros. Any operation that you perform on this data works on 40,001 fields that are not important.

The Greenplum Database built-in array datatype utilizes a bitmap for null values, but it is a poor choice for this use case because it is not optimized for float8[] or for long runs of zeros instead of nulls, and the bitmap is not run-length-encoding- (RLE) compressed. Even if each zero were stored as a NULL in the array, the bitmap for nulls would use 5KB to mark the nulls, which is not nearly as efficient as it could be.

The Greenplum Database gp_sparse_vector module defines a data type and a simple RLE-based scheme that is biased toward being efficient for zero value bitmaps. This scheme uses only 6 bytes for bitmap storage.

> **Note:** The sparse vector data type defined by the gp_sparse_vector module is named svec. svec supports only float8 vector values.

You can construct an svec directly from a float array as follows:

```
SELECT ('{0, 13, 37, 53, 0, 71 }'::float8[])::svec;
```

The gp_sparse_vector module supports the vector operators <, >, *, **, /, =, +, sum(), vec_count_nonzero(), and so on. These operators take advantage of the efficient sparse storage format, making computations on svecs faster.

The plus (+) operator adds each of the terms of two vectors of the same dimension together. For example, if vector a = {0,1,5} and vector b = {4,3,2}, you would compute the vector addition as follows:

```
SELECT ('{0,1,5}'::float8[]::svec + '{4,3,2}'::float8[]::svec)::float8[];
 float8
---------
 {4,4,7}
```

A vector dot product (`%*%`) between vectors a and b returns a scalar result of type `float8`. Compute the dot product (`(0*4+1*3+5*2)=13`) as follows:

```
SELECT '{0,1,5}'::float8[]::svec %*% '{4,3,2}'::float8[]::svec;
 ?column?
----------
       13
```

Special vector aggregate functions are also useful. `sum()` is self explanatory. `vec_count_nonzero()` evaluates the count of non-zero terms found in a set of `svec` and returns an `svec` with the counts. For instance, for the set of vectors `{0,1,5},{10,0,3},{0,0,3},{0,1,0}`, the count of non-zero terms would be `{1,2,3}`. Use `vec_count_nonzero()` to compute the count of these vectors:

```
CREATE TABLE listvecs( a svec );

INSERT INTO listvecs VALUES ('{0,1,5}'::float8[]),
    ('{10,0,3}'::float8[]),
    ('{0,0,3}'::float8[]),
    ('{0,1,0}'::float8[]);

SELECT vec_count_nonzero( a )::float8[] FROM listvecs;
 count_vec
-----------
 {1,2,3}
(1 row)
```

Additional Module Documentation

Refer to the `gp_sparse_vector` README*s* in the *Greenplum Database github repository* for additional information about this module.

Apache MADlib includes an extended implementation of sparse vectors. See the *MADlib Documentation* for a description of this MADlib module.

Example

A text classification example that describes a dictionary and some documents follows. You will create Greenplum Database tables representing a dictionary and some documents. You then perform document classification using vector arithmetic on word counts and proportions of dictionary words in each document.

Suppose that you have a dictionary composed of words in a text array. Create a table to store the dictionary data and insert some data (words) into the table. For example:

```
CREATE TABLE features (dictionary text[][]) DISTRIBUTED RANDOMLY;
INSERT INTO features
    VALUES
 ('{am,before,being,bothered,corpus,document,i,in,is,me,never,now,'
            'one,really,second,the,third,this,until}');
```

You have a set of documents, also defined as an array of words. Create a table to represent the documents and insert some data into the table:

```
CREATE TABLE documents(docnum int, document text[]) DISTRIBUTED RANDOMLY;
INSERT INTO documents VALUES
    (1,'{this,is,one,document,in,the,corpus}'),
    (2,'{i,am,the,second,document,in,the,corpus}'),
    (3,'{being,third,never,really,bothered,me,until,now}'),
    (4,'{the,document,before,me,is,the,third,document}');
```

Using the dictionary and document tables, find the dictionary words that are present in each document. To do this, you first prepare a *Sparse Feature Vector*, or SFV, for each document. An SFV is a vector of dimension *N*, where *N* is the number of dictionary words, and each SFV contains a count of each dictionary word in the document.

You can use the `gp_extract_feature_histogram()` function to create an SFV from a document. `gp_extract_feature_histogram()` outputs an `svec` for each document that contains the count of each of the dictionary words in the ordinal positions of the dictionary.

```
SELECT gp_extract_feature_histogram(
    (SELECT dictionary FROM features LIMIT 1), document)::float8[], document
        FROM documents ORDER BY docnum;

     gp_extract_feature_histogram        |                      document

-----------------------------------------
+-------------------------------------------------
 {0,0,0,0,1,1,0,1,1,0,0,0,1,0,0,1,0,1,0} |
 {this,is,one,document,in,the,corpus}
 {1,0,0,0,1,1,1,1,0,0,0,0,0,0,1,2,0,0,0} |
 {i,am,the,second,document,in,the,corpus}
 {0,0,1,1,0,0,0,0,0,1,1,1,0,1,0,0,1,0,1} |
 {being,third,never,really,bothered,me,until,now}
 {0,1,0,0,0,2,0,0,1,1,0,0,0,0,0,2,1,0,0} |
 {the,document,before,me,is,the,third,document}

SELECT * FROM features;
                                                 dictionary
-------------------------------------------------------------------------------------------
 {am,before,being,bothered,corpus,document,i,in,is,me,never,now,one,really,second,the,th
```

The SFV of the second document, "i am the second document in the corpus", is `{1,3*0,1,1,1,1,6*0,1,2}`. The word "am" is the first ordinate in the dictionary, and there is `1` instance of it in the SFV. The word "before" has no instances in the document, so its value is `0`; and so on.

`gp_extract_feature_histogram()` is very speed optimized - it is a single routine version of a hash join that processes large numbers of documents into their SFVs in parallel at the highest possible speeds.

For the next part of the processing, generate a sparse vector of the dictionary dimension (19). The vectors that you generate for each document are referred to as the *corpus*.

```
CREATE table corpus (docnum int, feature_vector svec) DISTRIBUTED RANDOMLY;

INSERT INTO corpus
    (SELECT docnum,
        gp_extract_feature_histogram(
            (select dictionary FROM features LIMIT 1), document) from
 documents);
```

Count the number of times each feature occurs at least once in all documents:

```
SELECT (vec_count_nonzero(feature_vector))::float8[] AS count_in_document
 FROM corpus;

            count_in_document
-----------------------------------------
 {1,1,1,1,2,3,1,2,2,2,1,1,1,1,1,3,2,1,1}
```

Count all occurrences of each term in all documents:

```
SELECT (sum(feature_vector))::float8[] AS sum_in_document FROM corpus;
```

```
              sum_in_document
----------------------------------------
 {1,1,1,1,2,4,1,2,2,2,1,1,1,1,1,5,2,1,1}
```

The remainder of the classification process is vector math. The count is turned into a weight that reflects *Term Frequency / Inverse Document Frequency* (tf/idf). The calculation for a given term in a given document is:

```
#_times_term_appears_in_this_doc * log( #_docs /
 #_docs_the_term_appears_in )
```

`#_docs` is the total number of documents (4 in this case). Note that there is one divisor for each dictionary word and its value is the number of times that word appears in the document.

For example, the term "document" in document 1 would have a weight of `1 * log( 4/3 )`. In document 4, the term would have a weight of `2 * log( 4/3 )`. Terms that appear in every document would have weight `0`.

This single vector for the whole corpus is then scalar product multiplied by each document SFV to produce the tf/idf.

Calculate the tf/idf:

```
SELECT docnum, (feature_vector*logidf)::float8[] AS tf_idf
    FROM (SELECT log(count(feature_vector)/
vec_count_nonzero(feature_vector)) AS logidf FROM corpus)
    AS foo, corpus ORDER BY docnum;
 docnum |
       tf_idf

--------
+-------------------------------------------------------------------------------------
      1 |
 {0,0,0,0,0.693147180559945,0.287682072451781,0,0.693147180559945,0.693147180559945,0,0,
      2 |
 {1.38629436111989,0,0,0,0.693147180559945,0.287682072451781,1.38629436111989,0.6931471
      3 |
 {0,0,1.38629436111989,1.38629436111989,0,0,0,0,0,0.693147180559945,1.38629436111989,1.3
}
      4 |
 {0,1.38629436111989,0,0,0,0.575364144903562,0,0,0.693147180559945,0.693147180559945,0,0
```

You can determine the *angular distance* between one document and the rest of the documents using the ACOS of the dot product of the document vectors:

```
CREATE TABLE weights AS
    (SELECT docnum, (feature_vector*logidf) tf_idf
        FROM (SELECT log(count(feature_vector)/
vec_count_nonzero(feature_vector))
       AS logidf FROM corpus) foo, corpus ORDER BY docnum)
    DISTRIBUTED RANDOMLY;
```

Calculate the angular distance between the first document and every other document:

```
SELECT docnum, trunc((180.*(ACOS(dmin(1.,(tf_idf%*%testdoc)/
(l2norm(tf_idf)*l2norm(testdoc))))/(4.*ATAN(1.))))::numeric,2)
     AS angular_distance FROM weights,
     (SELECT tf_idf testdoc FROM weights WHERE docnum = 1 LIMIT 1) foo
ORDER BY 1;

 docnum | angular_distance
--------+------------------
```

```
      1 |             0.00
      2 |            78.82
      3 |            90.00
      4 |            80.02
```

You can see that the angular distance between document 1 and itself is 0 degrees, and between document 1 and 3 is 90 degrees because they share no features at all.

hstore

The `hstore` module implements a data type for storing sets of (key,value) pairs within a single Greenplum Database data field. This can be useful in various scenarios, such as rows with many attributes that are rarely examined, or semi-structured data.

The Greenplum Database `hstore` module is equivalent to the PostgreSQL `hstore` module. There are no Greenplum Database or MPP-specific considerations for the module.

Installing and Registering the Module

The `hstore` module is installed when you install Greenplum Database. Before you can use any of the data types or functions defined in the module, you must register the `hstore` extension in each database in which you want to use the objects. Refer to *Installing Additional Supplied Modules* for more information.

Module Documentation

See *hstore* in the PostgreSQL documentation for detailed information about the data types and functions defined in this module.

orafce

The `orafce` module provides Oracle Compatibility SQL functions in Greenplum Database. These functions target PostgreSQL but can also be used in Greenplum.

The Greenplum Database `orafce` module is a modified version of the *open source Orafce PostgreSQL module extension*. The modified `orafce` source files for Greenplum Database can be found in the `gpcontrib/orafce` directory in the *Greenplum Database open source project*. The source reflects the Orafce 3.6.1 release and additional commits to *3af70a28f6*.

There are some restrictions and limitations when you use the module in Greenplum Database.

Installing and Registering the Module

> **Note:** Always use the Oracle Compatibility Functions module included with your Greenplum Database version. Before upgrading to a new Greenplum Database version, uninstall the compatibility functions from each of your databases, and then, when the upgrade is complete, reinstall the compatibility functions from the new Greenplum Database release. See the Greenplum Database release notes for upgrade prerequisites and procedures.

The `orafce` module is installed when you install Greenplum Database. Before you can use any of the functions defined in the module, you must register the `orafce` extension in each database in which you want to use the functions. Refer to *Installing Additional Supplied Modules* for more information.

Greenplum Database Considerations

The following functions are available by default in Greenplum Database and do not require installing the Oracle Compatibility Functions:

- sinh()
- tanh()

- cosh()
- decode() (See *Greenplum Implementation Differences* for more information.)

Greenplum Implementation Differences

There are differences in the implementation of the compatibility functions in Greenplum Database from the original PostgreSQL `orafce` module extension implementation. Some of the differences are as follows:

- The original `orafce` module implementation performs a decimal round off, the Greenplum Database implementation does not:
  - 2.00 becomes 2 in the original module implementation
  - 2.00 remains 2.00 in the Greenplum Database implementation
- The provided Oracle compatibility functions handle implicit type conversions differently. For example, using the `decode` function:

  ```
  decode(expression, value, return [,value, return]...
              [, default])
  ```

  The original `orafce` module implementation automatically converts *expression* and each *value* to the data type of the first *value* before comparing. It automatically converts *return* to the same data type as the first result.

  The Greenplum Database implementation restricts *return* and `default` to be of the same data type. The *expression* and *value* can be different types if the data type of *value* can be converted into the data type of the *expression*. This is done implicitly. Otherwise, `decode` fails with an `invalid input syntax` error. For example:

  ```
  SELECT decode('a','M',true,false);
  CASE
  ------
   f
  (1 row)
  SELECT decode(1,'M',true,false);
  ERROR: Invalid input syntax for integer:"M"
  LINE 1: SELECT decode(1,'M',true,false);
  ```

- Numbers in `bigint` format are displayed in scientific notation in the original `orafce` module implementation but not in the Greenplum Database implementation:
  - 9223372036854775 displays as 9.2234E+15 in the original implementation
  - 9223372036854775 remains 9223372036854775 in the Greenplum Database implementation
- The default date and timestamp format in the original `orafce` module implementation is different than the default format in the Greenplum Database implementation. If the following code is executed:

  ```
  CREATE TABLE TEST(date1 date, time1 timestamp, time2
                    timestamp with time zone);
  INSERT INTO TEST VALUES ('2001-11-11','2001-12-13
                   01:51:15','2001-12-13 01:51:15 -08:00');
  SELECT DECODE(date1, '2001-11-11', '2001-01-01') FROM TEST;
  ```

  The Greenplum Database implementation returns the row, but the original implementation returns no rows.

  Note: The correct syntax when using the original `orafce` implementation to return the row is:

  ```
  SELECT DECODE(to_char(date1, 'YYYY-MM-DD'), '2001-11-11',
                '2001-01-01') FROM TEST
  ```

- The functions in the Oracle Compatibility Functions `dbms_alert` package are not implemented for Greenplum Database.

- The `decode()` function is removed from the Greenplum Database Oracle Compatibility Functions. The Greenplum Database parser internally converts a `decode()` function call to a `CASE` statement.

Using orafce

Some Oracle Compatibility Functions reside in the `oracle` schema. To access them, set the search path for the database to include the `oracle` schema name. For example, this command sets the default search path for a database to include the `oracle` schema:

```
ALTER DATABASE db_name SET search_path = "$user", public, oracle;
```

Note the following differences when using the Oracle Compatibility Functions with PostgreSQL vs. using them with Greenplum Database:

- If you use validation scripts, the output may not be exactly the same as with the original `orafce` module implementation.
- The functions in the Oracle Compatibility Functions `dbms_pipe` package execute only on the Greenplum Database master host.
- The upgrade scripts in the Orafce project do not work with Greenplum Database.

Additional Module Documentation

Refer to the *README* and *Greenplum Database orafce documentation* in the Greenplum Database github repository for detailed information about the individual functions and supporting objects provided in this module.

pageinspect

The `pageinspect` module provides functions for low level inspection of the contents of database pages. `pageinspect` is available only to Greenplum Database superusers.

The Greenplum Database `pageinspect` module is based on the PostgreSQL `pageinspect` module. The Greenplum version of the module differs only in that it does not allow inspection of pages belonging to append-optimized or external relations.

Installing and Registering the Module

The `pageinspect` module is installed when you install Greenplum Database. Before you can use any of the functions defined in the module, you must register the `pageinspect` extension in each database in which you want to use the functions. Refer to *Installing Additional Supplied Modules* for more information.

Module Documentation

See *`pageinspect`* in the PostgreSQL documentation for detailed information about the individual functions in this module.

> **Note:** For `pageinspect` functions that read data from a database, the function reads data only from the segment instance where the function is executed. For example, the `get_raw_page()` function returns a `block number out of range` error when you try to read data from a user-defined table on the Greenplum Database master because there is no data in the table on the master segment. The function will read data from a system catalog table on the master segment.

pgcrypto

Greenplum Database is installed with an optional module of encryption/decryption functions called `pgcrypto`. The `pgcrypto` functions allow database administrators to store certain columns of data in encrypted form. This adds an extra layer of protection for sensitive data, as data stored in Greenplum

Database in encrypted form cannot be read by anyone who does not have the encryption key, nor can it be read directly from the disks.

> **Note:** The `pgcrypto` functions run inside the database server, which means that all the data and passwords move between `pgcrypto` and the client application in clear-text. For optimal security, consider also using SSL connections between the client and the Greenplum master server.

Installing and Registering the Module

The `pgcrypto` module is installed when you install Greenplum Database. Before you can use any of the functions defined in the module, you must register the `pgcrypto` extension in each database in which you want to use the functions. Refer to *Installing Additional Supplied Modules* for more information.

Additional Module Documentation

Refer to *pgcrypto* in the PostgreSQL documentation for more information about the individual functions in this module.

sslinfo

The `sslinfo` module provides information about the SSL certificate that the current client provided when connecting to Greenplum. Most functions in this module return NULL if the current connection does not use SSL.

The Greenplum Database `sslinfo` module is equivalent to the PostgreSQL `sslinfo` module. There are no Greenplum Database or MPP-specific considerations for the module.

Installing and Registering the Module

The `sslinfo` module is installed when you install Greenplum Database. Before you can use any of the functions defined in the module, you must register the `sslinfo` extension in each database in which you want to use the functions. Refer to *Installing Additional Supplied Modules* for more information.

Module Documentation

See *`sslinfo`* in the PostgreSQL documentation for detailed information about the individual functions in this module.

Character Set Support

The character set support in Greenplum Database allows you to store text in a variety of character sets, including single-byte character sets such as the ISO 8859 series and multiple-byte character sets such as EUC (Extended Unix Code), UTF-8, and Mule internal code. All supported character sets can be used transparently by clients, but a few are not supported for use within the server (that is, as a server-side encoding). The default character set is selected while initializing your Greenplum Database array using `gpinitsystem`. It can be overridden when you create a database, so you can have multiple databases each with a different character set.

Table 113: Greenplum Database Character Sets [10]

Name	Description	Language	Server?	Bytes/Char	Aliases
BIG5	Big Five	Traditional Chinese	No	1-2	WIN950, Windows950
EUC_CN	Extended UNIX Code-CN	Simplified Chinese	Yes	1-3	
EUC_JP	Extended UNIX Code-JP	Japanese	Yes	1-3	
EUC_KR	Extended UNIX Code-KR	Korean	Yes	1-3	
EUC_TW	Extended UNIX Code-TW	Traditional Chinese, Taiwanese	Yes	1-3	
GB18030	National Standard	Chinese	No	1-2	
GBK	Extended National Standard	Simplified Chinese	No	1-2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	Latin/Cyrillic	Yes	1	
ISO_8859_6	ISO 8859-6, ECMA 114	Latin/Arabic	Yes	1	
ISO_8859_7	ISO 8859-7, ECMA 118	Latin/Greek	Yes	1	
ISO_8859_8	ISO 8859-8, ECMA 121	Latin/Hebrew	Yes	1	
JOHAB	JOHA	Korean (Hangul)	Yes	1-3	
KOI8	KOI8-R(U)	Cyrillic	Yes	1	KOI8R
LATIN1	ISO 8859-1, ECMA 94	Western European	Yes	1	ISO88591

[10] **Not all APIs support all the listed character sets. For example, the JDBC driver does not support MULE_INTERNAL, LATIN6, LATIN8, and LATIN10.**

Name	Description	Language	Server?	Bytes/Char	Aliases
LATIN2	ISO 8859-2, ECMA 94	Central European	Yes	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	South European	Yes	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	North European	Yes	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	Turkish	Yes	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Nordic	Yes	1	ISO885910
LATIN7	ISO 8859-13	Baltic	Yes	1	ISO885913
LATIN8	ISO 8859-14	Celtic	Yes	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 with Euro and accents	Yes	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Romanian	Yes	1	ISO885916
MULE_ INTERNAL	Mule internal code	Multilingual Emacs	Yes	1-4	
SJIS	Shift JIS	Japanese	No	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SQL_ASCII	unspecified [11]	any	No	1	
UHC	Unified Hangul Code	Korean	No	1-2	WIN949, Windows949
UTF8	Unicode, 8-bit	all	Yes	1-4	Unicode
WIN866	Windows CP866	Cyrillic	Yes	1	ALT
WIN874	Windows CP874	Thai	Yes	1	
WIN1250	Windows CP1250	Central European	Yes	1	
WIN1251	Windows CP1251	Cyrillic	Yes	1	WIN
WIN1252	Windows CP1252	Western European	Yes	1	

[11] The SQL_ASCII setting behaves considerably differently from the other settings. Byte values 0-127 are interpreted according to the ASCII standard, while byte values 128-255 are taken as uninterpreted characters. If you are working with any non-ASCII data, it is unwise to use the SQL_ASCII setting as a client encoding. SQL_ASCII is not supported as a server encoding.

Name	Description	Language	Server?	Bytes/Char	Aliases
WIN1253	Windows CP1253	Greek	Yes	1	
WIN1254	Windows CP1254	Turkish	Yes	1	
WIN1255	Windows CP1255	Hebrew	Yes	1	
WIN1256	Windows CP1256	Arabic	Yes	1	
WIN1257	Windows CP1257	Baltic	Yes	1	
WIN1258	Windows CP1258	Vietnamese	Yes	1	ABC, TCVN, TCVN5712, VSCII

Setting the Character Set

gpinitsystem defines the default character set for a Greenplum Database system by reading the setting of the `ENCODING` parameter in the `gp_init_config` file at initialization time. The default character set is `UNICODE` or `UTF8`.

You can create a database with a different character set besides what is used as the system-wide default. For example:

```
=> CREATE DATABASE korean WITH ENCODING 'EUC_KR';
```

Important: Although you can specify any encoding you want for a database, it is unwise to choose an encoding that is not what is expected by the locale you have selected. The `LC_COLLATE` and `LC_CTYPE` settings imply a particular encoding, and locale-dependent operations (such as sorting) are likely to misinterpret data that is in an incompatible encoding.

Since these locale settings are frozen by gpinitsystem, the apparent flexibility to use different encodings in different databases is more theoretical than real.

One way to use multiple encodings safely is to set the locale to `C` or `POSIX` during initialization time, thus disabling any real locale awareness.

Character Set Conversion Between Server and Client

Greenplum Database supports automatic character set conversion between server and client for certain character set combinations. The conversion information is stored in the master *pg_conversion* system catalog table. Greenplum Database comes with some predefined conversions or you can create a new conversion using the SQL command `CREATE CONVERSION`.

Table 114: Client/Server Character Set Conversions

Server Character Set	Available Client Character Sets
BIG5	not supported as a server encoding
EUC_CN	EUC_CN, MULE_INTERNAL, UTF8
EUC_JP	EUC_JP, MULE_INTERNAL, SJIS, UTF8
EUC_KR	EUC_KR, MULE_INTERNAL, UTF8

Server Character Set	Available Client Character Sets
EUC_TW	EUC_TW, BIG5, MULE_INTERNAL, UTF8
GB18030	not supported as a server encoding
GBK	not supported as a server encoding
ISO_8859_5	ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	ISO_8859_6, UTF8
ISO_8859_7	ISO_8859_7, UTF8
ISO_8859_8	ISO_8859_8, UTF8
JOHAB	JOHAB, UTF8
KOI8	KOI8, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
LATIN1	LATIN1, MULE_INTERNAL, UTF8
LATIN2	LATIN2, MULE_INTERNAL, UTF8, WIN1250
LATIN3	LATIN3, MULE_INTERNAL, UTF8
LATIN4	LATIN4, MULE_INTERNAL, UTF8
LATIN5	LATIN5, UTF8
LATIN6	LATIN6, UTF8
LATIN7	LATIN7, UTF8
LATIN8	LATIN8, UTF8
LATIN9	LATIN9, UTF8
LATIN10	LATIN10, UTF8
MULE_INTERNAL	MULE_INTERNAL, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	not supported as a server encoding
SQL_ASCII	not supported as a server encoding
UHC	not supported as a server encoding
UTF8	all supported encodings
WIN866	WIN866
ISO_8859_5	KOI8, MULE_INTERNAL, UTF8, WIN1251
WIN874	WIN874, UTF8
WIN1250	WIN1250, LATIN2, MULE_INTERNAL, UTF8
WIN1251	WIN1251, ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866
WIN1252	WIN1252, UTF8
WIN1253	WIN1253, UTF8

Server Character Set	Available Client Character Sets
WIN1254	WIN1254, UTF8
WIN1255	WIN1255, UTF8
WIN1256	WIN1256, UTF8
WIN1257	WIN1257, UTF8
WIN1258	WIN1258, UTF8

To enable automatic character set conversion, you have to tell Greenplum Database the character set (encoding) you would like to use in the client. There are several ways to accomplish this:

- Using the `\encoding` command in `psql`, which allows you to change client encoding on the fly.
- Using `SETclient_encoding TO`.

  To set the client encoding, use the following SQL command:

  ```
  => SET CLIENT_ENCODING TO 'value';
  ```

  To query the current client encoding:

  ```
  => SHOW client_encoding;
  ```

  To return to the default encoding:

  ```
  => RESET client_encoding;
  ```

- Using the `PGCLIENTENCODING` environment variable. When `PGCLIENTENCODING` is defined in the client's environment, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)
- Setting the configuration parameter `client_encoding`. If `client_encoding` is set in the master `postgresql.conf` file, that client encoding is automatically selected when a connection to Greenplum Database is made. (This can subsequently be overridden using any of the other methods mentioned above.)

If the conversion of a particular character is not possible " suppose you chose `EUC_JP` for the server and `LATIN1` for the client, then some Japanese characters do not have a representation in `LATIN1` " then an error is reported.

If the client character set is defined as `SQL_ASCII`, encoding conversion is disabled, regardless of the server's character set. The use of `SQL_ASCII` is unwise unless you are working with all-ASCII data. `SQL_ASCII` is not supported as a server encoding.

Server Configuration Parameters

There are many Greenplum server configuration parameters that affect the behavior of the Greenplum Database system. Many of these configuration parameters have the same names, settings, and behaviors as in a regular PostgreSQL database system.

- *Parameter Types and Values* describes the parameter data types and values.
- *Setting Parameters* describes limitations on who can change them and where or when they can be set.
- *Parameter Categories* organizes parameters by functionality.
- *Configuration Parameters* lists the parameter descriptions in alphabetic order.

Parameter Types and Values

All parameter names are case-insensitive. Every parameter takes a value of one of the following types: `Boolean`, `integer`, `floating point`, `enum`, or `string`.

Boolean values may be specified as `ON`, `OFF`, `TRUE`, `FALSE`, `YES`, `NO`, `1`, `0` (all case-insensitive).

Enum-type parameters are specified in the same manner as string parameters, but are restricted to a limited set of values. Enum parameter values are case-insensitive.

Some settings specify a memory size or time value. Each of these has an implicit unit, which is either kilobytes, blocks (typically eight kilobytes), milliseconds, seconds, or minutes. Valid memory size units are `kB` (kilobytes), `MB` (megabytes), and `GB` (gigabytes). Valid time units are `ms` (milliseconds), `s` (seconds), `min` (minutes), `h` (hours), and `d` (days). Note that the multiplier for memory units is 1024, not 1000. A valid time expression contains a number and a unit. When specifying a memory or time unit using the `SET` command, enclose the value in quotes. For example:

```
SET statement_mem TO '200MB';
```

Note: There is no space between the value and the unit names.

Setting Parameters

Many of the configuration parameters have limitations on who can change them and where or when they can be set. For example, to change certain parameters, you must be a Greenplum Database superuser. Other parameters require a restart of the system for the changes to take effect. A parameter that is classified as *session* can be set at the system level (in the `postgresql.conf` file), at the database-level (using `ALTER DATABASE`), at the role-level (using `ALTER ROLE`), at the database- and role-level (`ALTER ROLE...IN DATABASE...SET`, or at the session-level (using `SET`). System parameters can only be set in the `postgresql.conf` file.

In Greenplum Database, the master and each segment instance has its own `postgresql.conf` file (located in their respective data directories). Some parameters are considered *local* parameters, meaning that each segment instance looks to its own `postgresql.conf` file to get the value of that parameter. You must set local parameters on every instance in the system (master and segments). Others parameters are considered *master* parameters. Master parameters need only be set at the master instance.

This table describes the values in the Settable Classifications column of the table in the description of a server configuration parameter.

Table 115: Settable Classifications

Set Classification	Description
master or local	A *master* parameter only needs to be set in the `postgresql.conf` file of the Greenplum master instance. The value for this parameter is then either passed to (or ignored by) the segments at run time. A local parameter must be set in the `postgresql.conf` file of the master AND each segment instance. Each segment instance looks to its own configuration to get the value for the parameter. Local parameters always requires a system restart for changes to take effect.
session or system	*Session* parameters can be changed on the fly within a database session, and can have a hierarchy of settings: at the system level (`postgresql.conf`), at the database level (`ALTER DATABASE...SET`), at the role level (`ALTER ROLE...SET`), at the database and role level (`ALTER ROLE...IN DATABASE...SET`), or at the session level (`SET`). If the parameter is set at multiple levels, then the most granular setting takes precedence (for example, session overrides database and role, database and role overrides role, role overrides database, and database overrides system). A *system* parameter can only be changed via the `postgresql.conf` file(s).
restart or reload	When changing parameter values in the postgresql.conf file(s), some require a *restart* of Greenplum Database for the change to take effect. Other parameter values can be refreshed by just reloading the server configuration file (using `gpstop -u`), and do not require stopping the system.
superuser	These session parameters can only be set by a database superuser. Regular database users cannot set this parameter.
read only	These parameters are not settable by database users or superusers. The current value of the parameter can be shown but not altered.

Parameter Categories

Configuration parameters affect categories of server behaviors, such as resource consumption, query tuning, and authentication. The following topics describe Greenplum configuration parameter categories.

- *Connection and Authentication Parameters*
- *System Resource Consumption Parameters*
- *GPORCA Parameters*
- *Query Tuning Parameters*

Connection and Authentication Parameters

These parameters control how clients connect and authenticate to Greenplum Database.

Connection Parameters

gp_connection_send_timeout
gp_vmem_idle_resource_timeout
listen_addresses
max_connections
max_prepared_transactions
superuser_reserved_connections
tcp_keepalives_count
tcp_keepalives_idle
tcp_keepalives_interval
unix_socket_directories
unix_socket_group
unix_socket_permissions

Security and Authentication Parameters

authentication_timeout
db_user_namespace
krb_caseins_users
krb_server_keyfile
password_encryption
password_hash_algorithm
ssl
ssl_ciphers

System Resource Consumption Parameters

These parameters set the limits for system resources consumed by Greenplum Database.

Memory Consumption Parameters

These parameters control system memory usage.

gp_vmem_idle_resource_timeout
gp_resource_group_memory_limit (resource group-based resource management)
gp_vmem_protect_limit (resource queue-based resource management)
gp_workfile_limit_per_query
gp_workfile_limit_per_segment
maintenance_work_mem
max_stack_depth

gp_vmem_protect_segworker_cache_limit
gp_workfile_limit_files_per_query
shared_buffers
temp_buffers

OS Resource Parameters

max_files_per_process
shared_preload_libraries

Cost-Based Vacuum Delay Parameters

> **Warning:** Do not use cost-based vacuum delay because it runs asynchronously among the segment instances. The vacuum cost limit and delay is invoked at the segment level without taking into account the state of the entire Greenplum Database array

You can configure the execution cost of `VACUUM` and `ANALYZE` commands to reduce the I/O impact on concurrent database activity. When the accumulated cost of I/O operations reaches the limit, the process performing the operation sleeps for a while, Then resets the counter and continues execution

vacuum_cost_delay
vacuum_cost_limit
vacuum_cost_page_dirty
vacuum_cost_page_hit
vacuum_cost_page_miss

Transaction ID Management Parameters

xid_stop_limit
xid_warn_limit

GPORCA Parameters

These parameters control the usage of GPORCA by Greenplum Database. For information about GPORCA, see *About GPORCA* in the *Greenplum Database Administrator Guide*.

gp_enable_relsize_collection
optimizer
optimizer_analyze_root_partition
optimizer_array_expansion_threshold
optimizer_cte_inlining_bound
optimizer_control
optimizer_enable_associativity
optimizer_enable_dml
optimizer_enable_master_only_queries
optimizer_force_agg_skew_avoidance
optimizer_force_multistage_agg
optimizer_force_three_stage_scalar_dqa
optimizer_join_arity_for_associativity_commutativity
optimizer_join_order
optimizer_join_order_threshold
optimizer_mdcache_size
optimizer_metadata_caching
optimizer_parallel_union
optimizer_penalize_skew
optimizer_print_missing_stats
optimizer_print_optimization_stats
optimizer_sort_factor
optimizer_use_gpdb_allocators

Query Tuning Parameters

These parameters control aspects of SQL query processing such as query operators and operator settings and statistics sampling.

Postgres Planner Control Parameters

The following parameters control the types of plan operations the Postgres Planner can use. Enable or disable plan operations to force the Postgres Planner to choose a different plan. This is useful for testing and comparing query performance using different plan types.

enable_bitmapscan
enable_groupagg
enable_hashagg
enable_hashjoin
enable_indexscan
enable_mergejoin
enable_nestloop
enable_seqscan
enable_sort
enable_tidscan
gp_enable_agg_distinct
gp_enable_agg_distinct_pruning
gp_enable_direct_dispatch
gp_enable_fast_sri
gp_enable_groupext_distinct_gather
gp_enable_groupext_distinct_pruning
gp_enable_multiphase_agg
gp_enable_predicate_propagation
gp_enable_preunique
gp_enable_relsize_collection
gp_enable_sort_distinct
gp_enable_sort_limit

Postgres Planner Costing Parameters

> **Warning:** Do not adjust these query costing parameters. They are tuned to reflect Greenplum Database hardware configurations and typical workloads. All of these parameters are related. Changing one without changing the others can have adverse affects on performance.

cpu_index_tuple_cost
cpu_operator_cost
cpu_tuple_cost
cursor_tuple_fraction
effective_cache_size
gp_motion_cost_per_row
gp_segments_for_planner
random_page_cost
seq_page_cost

Database Statistics Sampling Parameters

These parameters adjust the amount of data sampled by an `ANALYZE` operation. Adjusting these parameters affects statistics collection system-wide. You can configure statistics collection on particular tables and columns by using the `ALTER TABLESET STATISTICS` clause.

default_statistics_target

Sort Operator Configuration Parameters

gp_enable_sort_distinct

gp_enable_sort_limit

Aggregate Operator Configuration Parameters

gp_enable_agg_distinct
gp_enable_agg_distinct_pruning
gp_enable_multiphase_agg
gp_enable_preunique
gp_enable_groupext_distinct_gather
gp_enable_groupext_distinct_pruning
gp_workfile_compression

Join Operator Configuration Parameters

join_collapse_limit
gp_adjust_selectivity_for_outerjoins
gp_hashjoin_tuples_per_bucket
gp_statistics_use_fkeys
gp_workfile_compression

Other Postgres Planner Configuration Parameters

from_collapse_limit
gp_enable_predicate_propagation
gp_max_plan_size
gp_statistics_pullup_from_child_partition

Query Plan Execution

Control the query plan execution.

gp_max_slices
plan_cache_mode

Error Reporting and Logging Parameters

These configuration parameters control Greenplum Database logging.

Log Rotation

log_rotation_age
log_rotation_size
log_truncate_on_rotation

When to Log

client_min_messages
gp_interconnect_debug_retry_interval
log_error_verbosity
log_min_duration_statement
log_min_error_statement
log_min_messages
optimizer_minidump

What to Log

debug_pretty_print
debug_print_parse
debug_print_plan
debug_print_prelim_plan
debug_print_rewritten
debug_print_slice_table
log_autostats
log_connections
log_disconnections
log_dispatch_stats
log_duration
log_executor_stats
log_hostname
gp_log_interconnect
log_parser_stats
log_planner_stats
log_statement
log_statement_stats
log_timezone
gp_debug_linger
gp_log_format
gp_reraise_signal

System Monitoring Parameters

These configuration parameters control Greenplum Database data collection and notifications related to database monitoring.

Greenplum Performance Database

The following parameters configure the data collection agents that populate the `gpperfmon` database.

gp_enable_gpperfmon
gp_gpperfmon_send_interval
gpperfmon_log_alert_level
gpperfmon_port

Query Metrics Collection Parameters

These parameters enable and configure query metrics collection. When enabled, Greenplum Database saves metrics to shared memory during query execution. These metrics are used by Pivotal Greenplum Command Center, which is included with Pivotal's commercial version of Greenplum Database.

gp_enable_query_metrics
gp_instrument_shmem_size

Runtime Statistics Collection Parameters

These parameters control the server statistics collection feature. When statistics collection is enabled, you can access the statistics data using the *pg_stat* family of system catalog views.

stats_queue_level
track_activities
track_counts
update_process_title

Automatic Statistics Collection Parameters

When automatic statistics collection is enabled, you can run `ANALYZE` automatically in the same transaction as an `INSERT`, `UPDATE`, `DELETE`, `COPY` or `CREATE TABLE...AS SELECT` statement when a certain threshold of rows is affected (`on_change`), or when a newly generated table has no

statistics (`on_no_stats`). To enable this feature, set the following server configuration parameters in your Greenplum Database master `postgresql.conf` file and restart Greenplum Database:

gp_autostats_mode

gp_autostats_mode_in_functions

gp_autostats_on_change_threshold

log_autostats

> **Warning:** Depending on the specific nature of your database operations, automatic statistics collection can have a negative performance impact. Carefully evaluate whether the default setting of `on_no_stats` is appropriate for your system.

Client Connection Default Parameters

These configuration parameters set defaults that are used for client connections.

Statement Behavior Parameters

check_function_bodies

default_tablespace

default_transaction_deferrable

default_transaction_isolation

default_transaction_read_only

search_path

statement_timeout

temp_tablespaces

vacuum_freeze_min_age

Locale and Formatting Parameters

client_encoding

DateStyle

extra_float_digits

IntervalStyle

lc_collate

lc_ctype

lc_messages

lc_monetary

lc_numeric

lc_time

TimeZone

Other Client Default Parameters

dynamic_library_path

explain_pretty_print

local_preload_libraries

Lock Management Parameters

These configuration parameters set limits for locks and deadlocks.

deadlock_timeout

gp_enable global_deadlock_detector

gp_global_deadlock_detector_period

lock_timeout

max_locks_per_transaction

Resource Management Parameters (Resource Queues)

The following configuration parameters configure the Greenplum Database resource management feature (resource queues), query prioritization, memory utilization and concurrency control.

gp_resqueue_memory_policy
gp_resqueue_priority
gp_resqueue_priority_cpucores_per_segment
gp_resqueue_priority_sweeper_interval
gp_vmem_idle_resource_timeout
gp_vmem_protect_limit
gp_vmem_protect_segworker_cache_limit
max_resource_queues
max_resource_portals_per_transaction
max_statement_mem
resource_cleanup_gangs_on_wait
resource_select_only
runaway_detector_activation_percent
statement_mem
stats_queue_level
vmem_process_interrupt

Resource Management Parameters (Resource Groups)

The following parameters configure the Greenplum Database resource group workload management feature.

gp_resgroup_memory_policy
gp_resource_group_bypass
gp_resource_group_cpu_limit
gp_resource_group_memory_limit
gp_resource_group_queuing_timeout
gp_resource_manager
gp_vmem_idle_resource_timeout
gp_vmem_protect_segworker_cache_limit
max_statement_mem
memory_spill_ratio
runaway_detector_activation_percent
statement_mem
vmem_process_interrupt

External Table Parameters

The following parameters configure the external tables feature of Greenplum Database.

gp_external_enable_exec
gp_external_enable_filter_pushdown
gp_external_max_segs
gp_initial_bad_row_limit
gp_reject_percent_threshold
readable_external_table_timeout
writable_external_table_bufsize
verify_gpfdists_cert

Database Table Parameters

The following parameter configures default option settings for Greenplum Database tables.

gp_create_table_random_default_distribution
gp_default_storage_options
gp_enable_exchange_default_partition

gp_enable_segment_copy_checking

gp_use_legacy_hashops

Append-Optimized Table Parameters

The following parameters configure the append-optimized tables feature of Greenplum Database.

max_appendonly_tables

gp_add_column_inherits_table_setting

gp_appendonly_compaction

gp_appendonly_compaction_threshold

validate_previous_free_tid

Past Version Compatibility Parameters

The following parameters provide compatibility with older PostgreSQL and Greenplum Database versions. You do not need to change these parameters in Greenplum Database.

PostgreSQL

array_nulls

backslash_quote

escape_string_warning

regex_flavor

standard_conforming_strings

transform_null_equals

Greenplum Database

gp_ignore_error_table

Greenplum Database Array Configuration Parameters

The parameters in this topic control the configuration of the Greenplum Database array and its components: segments, master, distributed transaction manager, master mirror, and interconnect.

Interconnect Configuration Parameters

gp_interconnect_fc_method

gp_interconnect_proxy_addresses

gp_interconnect_queue_depth

gp_interconnect_setup_timeout

gp_interconnect_snd_queue_depth

gp_interconnect_type

gp_max_packet_size

Note: Greenplum Database supports only the UDPIFC (default) and TCP interconnect types.

Dispatch Configuration Parameters

gp_cached_segworkers_threshold

gp_enable_direct_dispatch

gp_segment_connect_timeout

gp_set_proc_affinity

Fault Operation Parameters

gp_set_read_only
gp_fts_probe_interval
gp_fts_probe_retries
gp_fts_probe_threadcount
gp_fts_probe_timeout
gp_fts_replication_attempt_count
gp_log_fts

Distributed Transaction Management Parameters

gp_max_local_distributed_cache

Read-Only Parameters

gp_command_count
gp_content
gp_dbid
gp_role
gp_session_id
gp_server_version
gp_server_version_num

Greenplum Mirroring Parameters for Master and Segments

These parameters control the configuration of the replication between Greenplum Database primary master and standby master.

max_slot_wal_keep_size
repl_catchup_within_range
replication_timeout
wait_for_replication_threshold
wal_keep_segments
wal_receiver_status_interval

Greenplum PL/Java Parameters

The parameters in this topic control the configuration of the Greenplum Database PL/Java language.

pljava_classpath
pljava_classpath_insecure
pljava_statement_cache_size
pljava_release_lingering_savepoints
pljava_vmoptions

XML Data Parameters

The parameters in this topic control the configuration of the Greenplum Database XML data type.

xmlbinary

xmloption

Configuration Parameters

Descriptions of the Greenplum Database server configuration parameters listed alphabetically.

- *application_name*
- *array_nulls*
- *authentication_timeout*
- *backslash_quote*
- *block_size*
- *bonjour_name*
- *check_function_bodies*
- *client_encoding*
- *client_min_messages*
- *cpu_index_tuple_cost*
- *cpu_operator_cost*
- *cpu_tuple_cost*
- *cursor_tuple_fraction*
- *data_checksums*
- *DateStyle*
- *db_user_namespace*
- *deadlock_timeout*
- *debug_assertions*
- *debug_pretty_print*
- *debug_print_parse*
- *debug_print_plan*
- *debug_print_prelim_plan*
- *debug_print_rewritten*
- *debug_print_slice_table*
- *default_statistics_target*
- *default_tablespace*
- *default_text_search_config*
- *default_transaction_deferrable*
- *default_transaction_isolation*
- *default_transaction_read_only*
- *dynamic_library_path*
- *effective_cache_size*
- *enable_bitmapscan*
- *enable_groupagg*
- *enable_hashagg*
- *enable_hashjoin*
- *enable_indexscan*
- *enable_mergejoin*
- *enable_nestloop*
- *enable_seqscan*
- *enable_sort*
- *enable_tidscan*
- *listen_addresses*
- *log_planner_stats*
- *log_rotation_age*
- *log_rotation_size*
- *log_statement*
- *log_statement_stats*
- *log_temp_files*
- *log_timezone*
- *log_truncate_on_rotation*
- *maintenance_work_mem*
- *max_appendonly_tables*
- *max_connections*
- *max_files_per_process*
- *max_function_args*
- *max_identifier_length*
- *max_index_keys*
- *max_locks_per_transaction*
- *max_prepared_transactions*
- *max_resource_portals_per_transaction*
- *max_resource_queues*
- *max_slot_wal_keep_size*
- *max_stack_depth*
- *max_statement_mem*
- *memory_spill_ratio*
- *optimizer*
- *optimizer_array_expansion_threshold*
- *optimizer_analyze_root_partition*
- *optimizer_control*
- *optimizer_cte_inlining_bound*
- *optimizer_enable_associativity*
- *optimizer_enable_dml*
- *optimizer_enable_master_only_queries*
- *optimizer_force_agg_skew_avoidance*
- *optimizer_force_multistage_agg*
- *optimizer_force_three_stage_scalar_dqa*
- *optimizer_join_arity_for_associativity_commutativity*
- *optimizer_join_order*
- *optimizer_join_order_threshold*
- *optimizer_mdcache_size*
- *optimizer_metadata_caching*
- *optimizer_minidump*
- *optimizer_nestloop_factor*

- escape_string_warning
- explain_pretty_print
- extra_float_digits
- from_collapse_limit
- gp_add_column_inherits_table_setting
- gp_adjust_selectivity_for_outerjoins
- gp_appendonly_compaction
- gp_appendonly_compaction_threshold
- gp_autostats_mode
- gp_autostats_mode_in_functions
- gp_autostats_on_change_threshold
- gp_cached_segworkers_threshold
- gp_command_count
- gp_connection_send_timeout
- gp_content
- gp_create_table_random_default_distribution
- gp_dbid
- gp_debug_linger
- gp_default_storage_options
- gp_dynamic_partition_pruning
- gp_enable_agg_distinct
- gp_enable_agg_distinct_pruning
- gp_enable_direct_dispatch
- gp_enable_exchange_default_partition
- gp_enable_fast_sri
- gp_enable_global_deadlock_detector
- gp_enable_gpperfmon
- gp_enable_groupext_distinct_gather
- gp_enable_groupext_distinct_pruning
- gp_enable_multiphase_agg
- gp_enable_predicate_propagation
- gp_enable_preunique
- gp_enable_query_metrics
- gp_enable_relsize_collection
- gp_enable_segment_copy_checking
- gp_enable_sort_distinct
- gp_enable_sort_limit
- gp_external_enable_exec
- gp_external_max_segs
- gp_external_enable_filter_pushdown
- gp_fts_probe_interval
- gp_fts_probe_retries
- gp_fts_probe_threadcount
- gp_fts_probe_timeout
- gp_fts_replication_attempt_count
- gp_global_deadlock_detector_period
- gp_gpperfmon_send_interval
- gp_hashjoin_tuples_per_bucket
- gp_resource_manager
- gp_use_legacy_hashops
- optimizer_parallel_union
- optimizer_penalize_skew
- optimizer_print_missing_stats
- optimizer_print_optimization_stats
- optimizer_sort_factor
- optimizer_use_gpdb_allocators
- plan_cache_mode
- password_encryption
- password_hash_algorithm
- pljava_classpath
- pljava_classpath_insecure
- pljava_statement_cache_size
- pljava_release_lingering_savepoints
- pljava_vmoptions
- port
- random_page_cost
- readable_external_table_timeout
- repl_catchup_within_range
- replication_timeout
- regex_flavor
- resource_cleanup_gangs_on_wait
- resource_select_only
- runaway_detector_activation_percent
- search_path
- seq_page_cost
- server_encoding
- server_version
- server_version_num
- shared_buffers
- shared_preload_libraries
- ssl
- ssl_ciphers
- standard_conforming_strings
- statement_mem
- statement_timeout
- stats_queue_level
- superuser_reserved_connections
- tcp_keepalives_count
- tcp_keepalives_idle
- tcp_keepalives_interval
- temp_buffers
- temp_tablespaces
- TimeZone
- timezone_abbreviations
- track_activity_query_size
- track_activities
- track_counts
- transaction_isolation
- transaction_read_only
- transform_null_equals

• *gp_vmem_idle_resource_timeout* • *gp_vmem_protect_limit* • *gp_vmem_protect_segworker_cache_limit* • *gp_workfile_compression* • *gp_workfile_limit_files_per_query* • *gp_workfile_limit_per_query* • *gp_workfile_limit_per_segment* • *gpperfmon_log_alert_level* • *gpperfmon_port* • *ignore_checksum_failure* • *integer_datetimes* • *IntervalStyle* • *join_collapse_limit* • *krb_caseins_users* • *krb_server_keyfile* • *lc_collate* • *lc_ctype* • *lc_messages* • *lc_monetary* • *lc_numeric* • *lc_time*	• *unix_socket_directories* • *unix_socket_group* • *unix_socket_permissions* • *update_process_title* • *vacuum_cost_delay* • *vacuum_cost_limit* • *vacuum_cost_page_dirty* • *vacuum_cost_page_hit* • *vacuum_cost_page_miss* • *vacuum_freeze_min_age* • *validate_previous_free_tid* • *verify_gpfdists_cert* • *vmem_process_interrupt* • *wait_for_replication_threshold* • *wal_keep_segments* • *wal_receiver_status_interval* • *writable_external_table_bufsize* • *xid_stop_limit* • *xid_warn_limit* • *xmlbinary* • *xmloption*

application_name

Sets the application name for a client session. For example, if connecting via `psql`, this will be set to `psql`. Setting an application name allows it to be reported in log messages and statistics views.

Value Range	Default	Set Classifications
string		master session reload

array_nulls

This controls whether the array input parser recognizes unquoted NULL as specifying a null array element. By default, this is on, allowing array values containing null values to be entered. Greenplum Database versions before 3.0 did not support null values in arrays, and therefore would treat NULL as specifying a normal array element with the string value 'NULL'.

Value Range	Default	Set Classifications
Boolean	on	master session reload

authentication_timeout

Maximum time to complete client authentication. This prevents hung clients from occupying a connection indefinitely.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	1min	local system restart

backslash_quote

This controls whether a quote mark can be represented by \' in a string literal. The preferred, SQL-standard way to represent a quote mark is by doubling it ('') but PostgreSQL has historically also accepted \'. However, use of \' creates security risks because in some client character set encodings, there are multibyte characters in which the last byte is numerically equivalent to ASCII \.

Value Range	Default	Set Classifications
on (allow \' always) off (reject always) safe_encoding (allow only if client encoding does not allow ASCII \ within a multibyte character)	safe_encoding	master session reload

block_size

Reports the size of a disk block.

Value Range	Default	Set Classifications
number of bytes	32768	read only

bonjour_name

Specifies the Bonjour broadcast name. By default, the computer name is used, specified as an empty string. This option is ignored if the server was not compiled with Bonjour support.

Value Range	Default	Set Classifications
string	unset	master system restart

check_function_bodies

When set to off, disables validation of the function body string during `CREATE FUNCTION`. Disabling validation is occasionally useful to avoid problems such as forward references when restoring function definitions from a dump.

Value Range	Default	Set Classifications
Boolean	on	master session reload

client_encoding

Sets the client-side encoding (character set). The default is to use the same as the database encoding. See *Supported Character Sets* in the PostgreSQL documentation.

Value Range	Default	Set Classifications
character set	UTF8	master session reload

client_min_messages

Controls which message levels are sent to the client. Each level includes all the levels that follow it. The later the level, the fewer messages are sent.

Value Range	Default	Set Classifications
DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 LOG NOTICE WARNING ERROR FATAL PANIC	NOTICE	master session reload

`INFO` level messages are always sent to the client.

cpu_index_tuple_cost

For the Postgres Planner, sets the estimate of the cost of processing each index row during an index scan. This is measured as a fraction of the cost of a sequential page fetch.

Value Range	Default	Set Classifications
floating point	0.005	master session reload

cpu_operator_cost

For the Postgres Planner, sets the estimate of the cost of processing each operator in a WHERE clause. This is measured as a fraction of the cost of a sequential page fetch.

Value Range	Default	Set Classifications
floating point	0.0025	master session reload

cpu_tuple_cost

For the Postgres Planner, Sets the estimate of the cost of processing each row during a query. This is measured as a fraction of the cost of a sequential page fetch.

Value Range	Default	Set Classifications
floating point	0.01	master session reload

cursor_tuple_fraction

Tells the Postgres Planner how many rows are expected to be fetched in a cursor query, thereby allowing the Postgres Planner to use this information to optimize the query plan. The default of 1 means all rows will be fetched.

Value Range	Default	Set Classifications
integer	1	master session reload

data_checksums

Reports whether checksums are enabled for heap data storage in the database system. Checksums for heap data are enabled or disabled when the database system is initialized and cannot be changed.

Heap data pages store heap tables, catalog tables, indexes, and database metadata. Append-optimized storage has built-in checksum support that is unrelated to this parameter.

Greenplum Database uses checksums to prevent loading data corrupted in the file system into memory managed by database processes. When heap data checksums are enabled, Greenplum Database computes and stores checksums on heap data pages when they are written to disk. When a page is retrieved from disk, the checksum is verified. If the verification fails, an error is generated and the page is not permitted to load into managed memory.

If the `ignore_checksum_failure` configuration parameter has been set to on, a failed checksum verification generates a warning, but the page is allowed to be loaded into managed memory. If the page is then updated, it is flushed to disk and replicated to the mirror. This can cause data corruption to propagate to the mirror and prevent a complete recovery. Because of the potential for data loss, the `ignore_checksum_failure` parameter should only be enabled when needed to recover data. See *ignore_checksum_failure* for more information.

Value Range	Default	Set Classifications
Boolean	on	read only

DateStyle

Sets the display format for date and time values, as well as the rules for interpreting ambiguous date input values. This variable contains two independent components: the output format specification and the input/output specification for year/month/day ordering.

Value Range	Default	Set Classifications
<format>, <date style> where: <format> is ISO, Postgres, SQL, or German <date style> is DMY, MDY, or YMD	ISO, MDY	master session reload

db_user_namespace

This enables per-database user names. If on, you should create users as *username@dbname*. To create ordinary global users, simply append @ when specifying the user name in the client.

Value Range	Default	Set Classifications
Boolean	off	local system restart

deadlock_timeout

The time to wait on a lock before checking to see if there is a deadlock condition. On a heavily loaded server you might want to raise this value. Ideally the setting should exceed your typical transaction time, so as to improve the odds that a lock will be released before the waiter decides to check for deadlock.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	1s	local system restart

debug_assertions

Turns on various assertion checks.

Value Range	Default	Set Classifications
Boolean	off	local system restart

debug_pretty_print

Indents debug output to produce a more readable but much longer output format. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	on	master session reload

debug_print_parse

For each executed query, prints the resulting parse tree. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	off	master session reload

debug_print_plan

For each executed query, prints the Greenplum parallel query execution plan. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	off	master session reload

debug_print_prelim_plan

For each executed query, prints the preliminary query plan. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	off	master session reload

debug_print_rewritten

For each executed query, prints the query rewriter output. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	off	master session reload

debug_print_slice_table

For each executed query, prints the Greenplum query slice plan. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	off	master session reload

default_statistics_target

Sets the default statistics sampling target (the number of values that are stored in the list of common values) for table columns that have not had a column-specific target set via `ALTER TABLE SET STATISTICS`. Larger values may improve the quality of the Postgres Planner estimates.

Value Range	Default	Set Classifications
0 > Integer > 10000	100	master session reload

default_tablespace

The default tablespace in which to create objects (tables and indexes) when a `CREATE` command does not explicitly specify a tablespace.

Value Range	Default	Set Classifications
name of a tablespace	unset	master session reload

default_text_search_config

Selects the text search configuration that is used by those variants of the text search functions that do not have an explicit argument specifying the configuration. See *Using Full Text Search* for further information. The built-in default is `pg_catalog.simple`, but `initdb` will initialize the configuration file with a setting that corresponds to the chosen `lc_ctype` locale, if a configuration matching that locale can be identified.

Value Range	Default	Set Classifications
The name of a text search configuration.	`pg_catalog.simple`	master session reload

default_transaction_deferrable

When running at the `SERIALIZABLE` isolation level, a deferrable read-only SQL transaction may be delayed before it is allowed to proceed. However, once it begins executing it does not incur any of the overhead required to ensure serializability; so serialization code will have no reason to force it to abort because of concurrent updates, making this option suitable for long-running read-only transactions.

This parameter controls the default deferrable status of each new transaction. It currently has no effect on read-write transactions or those operating at isolation levels lower than `SERIALIZABLE`. The default is `off`.

> **Note:** Setting `default_transaction_deferrable` to on has no effect in Greenplum Database. Only read-only, `SERIALIZABLE` transactions can be deferred. However, Greenplum Database does not support the `SERIALIZABLE` transaction isolation level. See *SET TRANSACTION*.

Value Range	Default	Set Classifications
Boolean	off	master session reload

default_transaction_isolation

Controls the default isolation level of each new transaction. Greenplum Database treats `read uncommitted` the same as `read committed`, and treats `serializable` the same as `repeatable read`.

Value Range	Default	Set Classifications
read committed read uncommitted repeatable read serializable	read committed	master session reload

default_transaction_read_only

Controls the default read-only status of each new transaction. A read-only SQL transaction cannot alter non-temporary tables.

Value Range	Default	Set Classifications
Boolean	off	master session reload

dynamic_library_path

If a dynamically loadable module needs to be opened and the file name specified in the `CREATE FUNCTION` or `LOAD` command does not have a directory component (i.e. the name does not contain a slash), the system will search this path for the required file. The compiled-in PostgreSQL package library directory is substituted for $libdir. This is where the modules provided by the standard PostgreSQL distribution are installed.

Value Range	Default	Set Classifications
a list of absolute directory paths separated by colons	$libdir	local system restart

effective_cache_size

Sets the assumption about the effective size of the disk cache that is available to a single query for the Postgres Planner. This is factored into estimates of the cost of using an index; a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. This parameter has no effect on the size of shared memory allocated by a Greenplum server instance, nor does it reserve kernel disk cache; it is used only for estimation purposes.

Set this parameter to a number of 32K blocks (for example, `512` for 16MB), or specify the size of the effective cache (for example, `'32MB'` for 1024 blocks). The `gpconfig` utility and `SHOW` command display the effective cache size value in units such as 'MB' or 'kB'.

Value Range	Default	Set Classifications
floating point	512 (16GB)	master session reload

enable_bitmapscan

Enables or disables the use of bitmap-scan plan types by the Postgres Planner. Note that this is different than a Bitmap Index Scan. A Bitmap Scan means that indexes will be dynamically converted to bitmaps in memory when appropriate, giving faster index performance on complex queries against very large tables. It is used when there are multiple predicates on different indexed columns. Each bitmap per column can be compared to create a final list of selected tuples.

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_groupagg

Enables or disables the use of group aggregation plan types by the Postgres Planner.

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_hashagg

Enables or disables the use of hash aggregation plan types by the Postgres Planner.

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_hashjoin

Enables or disables the use of hash-join plan types by the Postgres Planner.

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_indexscan

Enables or disables the use of index-scan plan types by the Postgres Planner.

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_mergejoin

Enables or disables the use of merge-join plan types by the Postgres Planner. Merge join is based on the idea of sorting the left- and right-hand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the 'same place' in the sort order. In practice this means that the join operator must behave like equality.

Value Range	Default	Set Classifications
Boolean	off	master session reload

enable_nestloop

Enables or disables the use of nested-loop join plans by the Postgres Planner. It's not possible to suppress nested-loop joins entirely, but turning this variable off discourages the Postgres Planner from using one if there are other methods available.

Value Range	Default	Set Classifications
Boolean	off	master session reload

enable_seqscan

Enables or disables the use of sequential scan plan types by the Postgres Planner. It's not possible to suppress sequential scans entirely, but turning this variable off discourages the Postgres Planner from using one if there are other methods available.

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_sort

Enables or disables the use of explicit sort steps by the Postgres Planner. It's not possible to suppress explicit sorts entirely, but turning this variable off discourages the Postgres Planner from using one if there are other methods available.

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_tidscan

Enables or disables the use of tuple identifier (TID) scan plan types by the Postgres Planner.

Value Range	Default	Set Classifications
Boolean	on	master session reload

escape_string_warning

When on, a warning is issued if a backslash (\) appears in an ordinary string literal ('...' syntax). Escape string syntax (E'...') should be used for escapes, because in future versions, ordinary strings will have the SQL standard-conforming behavior of treating backslashes literally.

Value Range	Default	Set Classifications
Boolean	on	master session reload

explain_pretty_print

Determines whether EXPLAIN VERBOSE uses the indented or non-indented format for displaying detailed query-tree dumps.

Value Range	Default	Set Classifications
Boolean	on	master session reload

extra_float_digits

Adjusts the number of digits displayed for floating-point values, including float4, float8, and geometric data types. The parameter value is added to the standard number of digits. The value can be set as high as 3, to include partially-significant digits; this is especially useful for dumping float data that needs to be restored exactly. Or it can be set negative to suppress unwanted digits.

Value Range	Default	Set Classifications
integer	0	master session reload

from_collapse_limit

The Postgres Planner will merge sub-queries into upper queries if the resulting FROM list would have no more than this many items. Smaller values reduce planning time but may yield inferior query plans.

Value Range	Default	Set Classifications
1-*n*	20	master session reload

gp_add_column_inherits_table_setting

When adding a column to an append-optimized, column-oriented table with the *ALTER TABLE* command, this parameter controls whether the table's data compression parameters for a column (`compresstype`, `compresslevel`, and `blocksize`) can be inherited from the table values. The default is `off`, the table's data compression settings are not considered when adding a column to the table. If the value is `on`, the table's settings are considered.

When you create an append-optimized column-oriented table, you can set the table's data compression parameters `compresstype`, `compresslevel`, and `blocksize` for the table in the `WITH` clause. When you add a column, Greenplum Database sets each data compression parameter based on one of the following settings, in order of preference.

1. The data compression setting specified in the `ALTER TABLE` command `ENCODING` clause.
2. If this server configuration parameter is set to on, the table's data compression setting specified in the `WITH` clause when the table was created. Otherwise, the table's data compression setting is ignored.
3. The data compression setting specified in the server configuration parameter *gp_default_storage_options*.
4. The default data compression setting.

For information about the data storage compression parameters, see *CREATE TABLE*.

Value Range	Default	Set Classifications
Boolean	off	master session reload

gp_adjust_selectivity_for_outerjoins

Enables the selectivity of NULL tests over outer joins.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_appendonly_compaction

Enables compacting segment files during `VACUUM` commands. When disabled, `VACUUM` only truncates the segment files to the EOF value, as is the current behavior. The administrator may want to disable compaction in high I/O load situations or low space situations.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_appendonly_compaction_threshold

Specifies the threshold ratio (as a percentage) of hidden rows to total rows that triggers compaction of the segment file when `VACUUM` is run without the `FULL` option (a lazy vacuum). If the ratio of hidden rows in a segment file on a segment is less than this threshold, the segment file is not compacted, and a log message is issued.

Value Range	Default	Set Classifications
integer (%)	10	master session reload

gp_autostats_mode

Specifies the mode for triggering automatic statistics collection with ANALYZE. The on_no_stats option triggers statistics collection for CREATE TABLE AS SELECT, INSERT, or COPY operations on any table that has no existing statistics.

The on_change option triggers statistics collection only when the number of rows affected exceeds the threshold defined by gp_autostats_on_change_threshold. Operations that can trigger automatic statistics collection with on_change are:

CREATE TABLE AS SELECT

UPDATE

DELETE

INSERT

COPY

Default is on_no_stats.

> **Note:** For partitioned tables, automatic statistics collection is not triggered if data is inserted from the top-level parent table of a partitioned table.
>
> Automatic statistics collection is triggered if data is inserted directly in a leaf table (where the data is stored) of the partitioned table. Statistics are collected only on the leaf table.

Value Range	Default	Set Classifications
none	on_no_ stats	master
on_change		session
on_no_stats		reload

gp_autostats_mode_in_functions

Specifies the mode for triggering automatic statistics collection with ANALYZE for statements in procedural language functions. The none option disables statistics collection. The on_no_stats option triggers statistics collection for CREATE TABLE AS SELECT, INSERT, or COPY operations that are executed in functions on any table that has no existing statistics.

The on_change option triggers statistics collection only when the number of rows affected exceeds the threshold defined by gp_autostats_on_change_threshold. Operations in functions that can trigger automatic statistics collection with on_change are:

CREATE TABLE AS SELECT

UPDATE

DELETE

INSERT

COPY

Value Range	Default	Set Classifications
none	none	master
on_change		session
on_no_stats		reload

gp_autostats_on_change_threshold

Specifies the threshold for automatic statistics collection when `gp_autostats_mode` is set to `on_change`. When a triggering table operation affects a number of rows exceeding this threshold, `ANALYZE` is added and statistics are collected for the table.

Value Range	Default	Set Classifications
integer	2147483647	master session reload

gp_cached_segworkers_threshold

When a user starts a session with Greenplum Database and issues a query, the system creates groups or 'gangs' of worker processes on each segment to do the work. After the work is done, the segment worker processes are destroyed except for a cached number which is set by this parameter. A lower setting conserves system resources on the segment hosts, but a higher setting may improve performance for power-users that want to issue many complex queries in a row.

Value Range	Default	Set Classifications
integer > 0	5	master session reload

gp_command_count

Shows how many commands the master has received from the client. Note that a single SQLcommand might actually involve more than one command internally, so the counter may increment by more than one for a single query. This counter also is shared by all of the segment processes working on the command.

Value Range	Default	Set Classifications
integer > 0	1	read only

gp_connection_send_timeout

Timeout for sending data to unresponsive Greenplum Database user clients during query processing. A value of 0 disables the timeout, Greenplum Database waits indefinitely for a client. When the timeout is reached, the query is cancelled with this message:

```
Could not send data to client: Connection timed out.
```

Value Range	Default	Set Classifications
number of seconds	3600 (1 hour)	master system reload

gp_content

The local content id if a segment.

Value Range	Default	Set Classifications
integer		read only

gp_create_table_random_default_distribution

Controls table creation when a Greenplum Database table is created with a `CREATE TABLE` or `CREATE TABLE AS` command that does not contain a `DISTRIBUTED BY` clause.

For `CREATE TABLE`, if the value of the parameter is `off` (the default), and the table creation command does not contain a `DISTRIBUTED BY` clause, Greenplum Database chooses the table distribution key based on the command:

- If a `LIKE` or `INHERITS` clause is specified, then Greenplum copies the distribution key from the source or parent table.
- If a `PRIMARY KEY` or `UNIQUE` constraints are specified, then Greenplum chooses the largest subset of all the key columns as the distribution key.
- If neither constraints nor a `LIKE` or `INHERITS` clause is specified, then Greenplum chooses the first suitable column as the distribution key. (Columns with geometric or user-defined data types are not eligible as Greenplum distribution key columns.)

If the value of the parameter is set to `on`, Greenplum Database follows these rules to create a table when the `DISTRIBUTED BY` clause is not specified:

- If `PRIMARY KEY` or `UNIQUE` columns are not specified, the distribution of the table is random (`DISTRIBUTED RANDOMLY`). Table distribution is random even if the table creation command contains the `LIKE` or `INHERITS` clause.
- If `PRIMARY KEY` or `UNIQUE` columns are specified, a `DISTRIBUTED BY` clause must also be specified. If a `DISTRIBUTED BY` clause is not specified as part of the table creation command, the command fails.

For a `CREATE TABLE AS` command that does not contain a distribution clause:

- If the Postgres Planner creates the table, and the value of the parameter is `off`, the table distribution policy is determined based on the command.
- If the Postgres Planner creates the table, and the value of the parameter is `on`, the table distribution policy is random.
- If GPORCA creates the table, the table distribution policy is random. The parameter value has no affect.

For information about the Postgres Planner and GPORCA, see "Querying Data" in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
boolean	off	master system reload

gp_dbid

The local content dbid if a segment.

Value Range	Default	Set Classifications
integer		read only

gp_debug_linger

Number of seconds for a Greenplum process to linger after a fatal internal error.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	0	master session reload

gp_default_storage_options

Set the default values for the following table storage options when a table is created with the `CREATE TABLE` command.

- `appendoptimized`

  Note: You use the `appendoptimized=`*`value`* syntax to specify the append-optimized table storage type. `appendoptimized` is a thin alias for the `appendonly` legacy storage option. Greenplum Database stores `appendonly` in the catalog, and displays the same when listing the storage options for append-optimized tables.
- `blocksize`
- `checksum`
- `compresstype`
- `compresslevel`
- `orientation`

Specify multiple storage option values as a comma separated list.

You can set the storage options with this parameter instead of specifying the table storage options in the `WITH` of the `CREATE TABLE` command. The table storage options that are specified with the `CREATE TABLE` command override the values specified by this parameter.

Not all combinations of storage option values are valid. If the specified storage options are not valid, an error is returned. See the `CREATE TABLE` command for information about table storage options.

The defaults can be set for a database and user. If the server configuration parameter is set at different levels, this the order of precedence, from highest to lowest, of the table storage values when a user logs into a database and creates a table:

1. The values specified in a `CREATE TABLE` command with the `WITH` clause or `ENCODING` clause
2. The value of `gp_default_storage_options` that set for the user with the `ALTER ROLE...SET` command
3. The value of `gp_default_storage_options` that is set for the database with the `ALTER DATABASE...SET` command
4. The value of `gp_default_storage_options` that is set for the Greenplum Database system with the `gpconfig` utility

The parameter value is not cumulative. For example, if the parameter specifies the `appendoptimized` and `compresstype` options for a database and a user logs in and sets the parameter to specify the value for the `orientation` option, the `appendoptimized`, and `compresstype` values set at the database level are ignored.

This example `ALTER DATABASE` command sets the default `orientation` and `compresstype` table storage options for the database `mytest`.

```
ALTER DATABASE mytest SET gp_default_storage_options = 'orientation=column,
 compresstype=rle_type'
```

To create an append-optimized table in the `mytest` database with column-oriented table and RLE compression. The user needs to specify only `appendoptimized=TRUE` in the `WITH` clause.

This example `gpconfig` utility command sets the default storage option for a Greenplum Database system. If you set the defaults for multiple table storage options, the value must be enclosed in single quotes.

```
gpconfig -c 'gp_default_storage_options' -v 'appendoptimized=true,
 orientation=column'
```

This example `gpconfig` utility command shows the value of the parameter. The parameter value must be consistent across the Greenplum Database master and all segments.

```
gpconfig -s 'gp_default_storage_options'
```

Value Range	Default	Set Classifications [1]

`appendoptimized=` `TRUE` | `FALSE`
`blocksize=` integer between 8192 and 2097152
`checksum=` `TRUE` | `FALSE`
`compresstype=` `ZLIB` | `ZSTD` | `QUICKLZ`[2] | `RLE_TYPE` | `NONE`
`compresslevel=` integer between 0 and 19
`orientation=` `ROW` | `COLUMN`
	`appendoptimized=FALSE`
	`blocksize=32768`
	`checksum=TRUE`
	`compresstype=none`
	`compresslevel=0`
	`orientation=ROW`
		master
		session
		reload

Note: [1]The set classification when the parameter is set at the system level with the `gpconfig` utility.

Note: [2]QuickLZ compression is available only in the commercial release of Pivotal Greenplum Database.

gp_dynamic_partition_pruning

Enables plans that can dynamically eliminate the scanning of partitions.

Value Range	Default	Set Classifications
on/off	on	master session reload

gp_enable_agg_distinct

Enables or disables two-phase aggregation to compute a single distinct-qualified aggregate. This applies only to subqueries that include a single distinct-qualified aggregate function.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_agg_distinct_pruning

Enables or disables three-phase aggregation and join to compute distinct-qualified aggregates. This applies only to subqueries that include one or more distinct-qualified aggregate functions.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_direct_dispatch

Enables or disables the dispatching of targeted query plans for queries that access data on a single segment. When on, queries that target rows on a single segment will only have their query plan dispatched to that segment (rather than to all segments). This significantly reduces the response time of qualifying queries as there is no interconnect setup involved. Direct dispatch does require more CPU utilization on the master.

Value Range	Default	Set Classifications
Boolean	on	master system restart

gp_enable_exchange_default_partition

Controls availability of the `EXCHANGE DEFAULT PARTITION` clause for `ALTER TABLE`. The default value for the parameter is `off`. The clause is not available and Greenplum Database returns an error if the clause is specified in an `ALTER TABLE` command.

If the value is `on`, Greenplum Database returns a warning stating that exchanging the default partition might result in incorrect results due to invalid data in the default partition.

> **Warning:** Before you exchange the default partition, you must ensure the data in the table to be exchanged, the new default partition, is valid for the default partition. For example, the data in the new default partition must not contain data that would be valid in other leaf child partitions of the partitioned table. Otherwise, queries against the partitioned table with the exchanged default partition that are executed by GPORCA might return incorrect results.

Value Range	Default	Set Classifications
Boolean	off	master session reload

gp_enable_fast_sri

When set to `on`, the Postgres Planner plans single row inserts so that they are sent directly to the correct segment instance (no motion operation required). This significantly improves performance of single-row-insert statements.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_global_deadlock_detector

Controls whether the Greenplum Database Global Deadlock Detector is enabled to manage concurrent `UPDATE` and `DELETE` operations on heap tables to improve performance. See *Global Deadlock Detector* in the *Greenplum Database Administrator Guide*. The default is `off`, the Global Deadlock Detector is disabled.

If the Global Deadlock Detector is disabled (the default), Greenplum Database executes concurrent update and delete operations on a heap table serially.

If the Global Deadlock Detector is enabled, concurrent updates are permitted and the Global Deadlock Detector determines when a deadlock exists, and breaks the deadlock by cancelling one or more backend processes associated with the youngest transaction(s) involved.

Value Range	Default	Set Classifications
Boolean	off	master system restart

gp_enable_gpperfmon

Enables or disables the data collection agents that populate the `gpperfmon` database.

Value Range	Default	Set Classifications
Boolean	off	local system restart

gp_enable_groupext_distinct_gather

Enables or disables gathering data to a single node to compute distinct-qualified aggregates on grouping extension queries. When this parameter and `gp_enable_groupext_distinct_pruning` are both enabled, the Postgres Planner uses the cheaper plan.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_groupext_distinct_pruning

Enables or disables three-phase aggregation and join to compute distinct-qualified aggregates on grouping extension queries. Usually, enabling this parameter generates a cheaper query plan that the Postgres Planner will use in preference to existing plan.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_multiphase_agg

Enables or disables the use of two or three-stage parallel aggregation plans Postgres Planner. This approach applies to any subquery with aggregation. If `gp_enable_multiphase_agg` is off, then `gp_enable_agg_distinct` and `gp_enable_agg_distinct_pruning` are disabled.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_predicate_propagation

When enabled, the Postgres Planner applies query predicates to both table expressions in cases where the tables are joined on their distribution key column(s). Filtering both tables prior to doing the join (when possible) is more efficient.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_preunique

Enables two-phase duplicate removal for `SELECT DISTINCT` queries (not `SELECT COUNT(DISTINCT)`). When enabled, it adds an extra `SORT DISTINCT` set of plan nodes before motioning. In cases where the distinct operation greatly reduces the number of rows, this extra `SORT DISTINCT` is much cheaper than the cost of sending the rows across the Interconnect.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_query_metrics

Enables collection of query metrics. When query metrics collection is enabled, Greenplum Database collects metrics during query execution. The default is off.

After changing this configuration parameter, Greenplum Database must be restarted for the change to take effect.

The Greenplum Database metrics collection extension, when enabled, sends the collected metrics over UDP to a Pivotal Greenplum Command Center agent[1].

> **Note:** [1] The metrics collection extension is included in Pivotal's commercial version of Greenplum Database. Pivotal Greenplum Command Center is supported only with Pivotal Greenplum Database.

Value Range	Default	Set Classifications
Boolean	off	master system restart

gp_enable_relsize_collection

Enables GPORCA and the Postgres Planner to use the estimated size of a table (`pg_relation_size` function) if there are no statistics for the table. By default, GPORCA and the planner use a default value to estimate the number of rows if statistics are not available. The default behavior improves query optimization time and reduces resource queue usage in heavy workloads, but can lead to suboptimal plans.

This parameter is ignored for a root partition of a partitioned table. When GPORCA is enabled and the root partition does not have statistics, GPORCA always uses the default value. You can use `ANALZYE ROOTPARTITION` to collect statistics on the root partition. See *ANALYZE*.

Value Range	Default	Set Classifications
Boolean	off	master session reload

gp_enable_segment_copy_checking

Controls whether the distribution policy for a table (from the table `DISTRIBUTED` clause) is checked when data is copied into the table with the `COPY FROM...ON SEGMENT` command. If true, an error is returned if a row of data violates the distribution policy for a segment instance. The default is `true`.

If the value is `false`, the distribution policy is not checked. The data added to the table might violate the table distribution policy for the segment instance. Manual redistribution of table data might be required. See the `ALTER TABLE` clause `WITH REORGANIZE`.

The parameter can be set for a database system or a session. The parameter cannot be set for a specific database.

Value Range	Default	Set Classifications
Boolean	true	master session reload

gp_enable_sort_distinct

Enable duplicates to be removed while sorting.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_sort_limit

Enable `LIMIT` operation to be performed while sorting. Sorts more efficiently when the plan requires the first *limit_number* of rows at most.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_external_enable_exec

Enables or disables the use of external tables that execute OS commands or scripts on the segment hosts (`CREATE EXTERNAL TABLE EXECUTE` syntax). Must be enabled if using the Command Center or MapReduce features.

Value Range	Default	Set Classifications
Boolean	on	master system restart

gp_external_max_segs

Sets the number of segments that will scan external table data during an external table operation, the purpose being not to overload the system with scanning data and take away resources from other concurrent operations. This only applies to external tables that use the `gpfdist://` protocol to access external table data.

Value Range	Default	Set Classifications
integer	64	master session reload

gp_external_enable_filter_pushdown

Enable filter pushdown when reading data from external tables. If pushdown fails, a query is executed without pushing filters to the external data source (instead, Greenplum Database applies the same constraints to the result). See *Defining External Tables* for more information.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_fts_probe_interval

Specifies the polling interval for the fault detection process (`ftsprobe`). The `ftsprobe` process will take approximately this amount of time to detect a segment failure.

Value Range	Default	Set Classifications
10 - 3600 seconds	1min	master system restart

gp_fts_probe_retries

Specifies the number of times the fault detection process (`ftsprobe`) attempts to connect to a segment before reporting segment failure.

Value Range	Default	Set Classifications
integer	5	master system restart

gp_fts_probe_threadcount

Specifies the number of `ftsprobe` threads to create. This parameter should be set to a value equal to or greater than the number of segments per host.

Value Range	Default	Set Classifications
1 - 128	16	master system restart

gp_fts_probe_timeout

Specifies the allowed timeout for the fault detection process (`ftsprobe`) to establish a connection to a segment before declaring it down.

Value Range	Default	Set Classifications
10 - 3600 seconds	20 secs	master system restart

gp_fts_replication_attempt_count

Specifies the maximum number of times that Greenplum Database attempts to establish a primary-mirror replication connection. When this count is exceeded, the fault detection process (`ftsprobe`) stops retrying and marks the mirror down.

Value Range	Default	Set Classifications
0 - 100	10	master system reload

gp_global_deadlock_detector_period

Specifies the executing interval (in seconds) of the global deadlock detector background worker process.

Value Range	Default	Set Classifications
5 - `INT_MAX` secs	120 secs	master system reload

gp_log_fts

Controls the amount of detail the fault detection process (`ftsprobe`) writes to the log file.

Value Range	Default	Set Classifications
OFF TERSE VERBOSE DEBUG	TERSE	master system restart

gp_log_interconnect

Controls the amount of information that is written to the log file about communication between Greenplum Database segment instance worker processes. The default value is `terse`. The log information is written to both the master and segment instance logs.

Increasing the amount of logging could affect performance and increase disk space usage.

Value Range	Default	Set Classifications
off terse verbose debug	terse	master session reload

gp_log_gang

Controls the amount of information that is written to the log file about query worker process creation and query management. The default value is `OFF`, do not log information.

Value Range	Default	Set Classifications
OFF TERSE VERBOSE DEBUG	OFF	master session restart

gp_gpperfmon_send_interval

Sets the frequency that the Greenplum Database server processes send query execution updates to the data collection agent processes used to populate the `gpperfmon` database. Query operations executed during this interval are sent through UDP to the segment monitor agents. If you find that an excessive number of UDP packets are dropped during long-running, complex queries, you may consider increasing this value.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	1sec	master system restart superuser

gpperfmon_log_alert_level

Controls which message levels are written to the gpperfmon log. Each level includes all the levels that follow it. The later the level, the fewer messages are sent to the log.

> **Note:** If the `gpperfmon` database is installed and is monitoring the database, the default value is warning.

Value Range	Default	Set Classifications
none warning error fatal panic	none	local system restart

gp_hashjoin_tuples_per_bucket

Sets the target density of the hash table used by HashJoin operations. A smaller value will tend to produce larger hash tables, which can increase join performance.

Value Range	Default	Set Classifications
integer	5	master session reload

gp_ignore_error_table

Controls Greenplum Database behavior when the deprecated `INTO ERROR TABLE` clause is specified in a `CREATE EXTERNAL TABLE` or `COPY` command.

> **Note:** The `INTO ERROR TABLE` clause was deprecated and removed in Greenplum Database 5. In Greenplum Database 7, this parameter will be removed as well, causing all `INTO ERROR TABLE` invocations be yield a syntax error.

The default value is `false`, Greenplum Database returns an error if the `INTO ERROR TABLE` clause is specified in a command.

If the value is `true`, Greenplum Database ignores the clause, issues a warning, and executes the command without the `INTO ERROR TABLE` clause. In Greenplum Database 5.x and later, you access the error log information with built-in SQL functions. See the *`CREATE EXTERNAL TABLE`* or *`COPY`* command.

You can set this value to `true` to avoid the Greenplum Database error when you run applications that execute `CREATE EXTERNAL TABLE` or `COPY` commands that include the Greenplum Database 4.3.x `INTO ERROR TABLE` clause.

Value Range	Default	Set Classifications
Boolean	false	master session reload

gp_initial_bad_row_limit

For the parameter value *n*, Greenplum Database stops processing input rows when you import data with the `COPY` command or from an external table if the first *n* rows processed contain formatting errors. If a valid row is processed within the first *n* rows, Greenplum Database continues processing input rows.

Setting the value to 0 disables this limit.

The `SEGMENT REJECT LIMIT` clause can also be specified for the `COPY` command or the external table definition to limit the number of rejected rows.

`INT_MAX` is the largest value that can be stored as an integer on your system.

Value Range	Default	Set Classifications
integer 0 - `INT_MAX`	1000	master session reload

gp_instrument_shmem_size

The amount of shared memory, in kilobytes, allocated for query metrics. The default is 5120 and the maximum is 131072. At startup, if *gp_enable_query_metrics* is set to on, Greenplum Database allocates space in shared memory to save query metrics. This memory is organized as a header and a list of slots. The number of slots needed depends on the number of concurrent queries and the number of execution plan nodes per query. The default value, 5120, is based on a Greenplum Database system that executes a maximum of about 250 concurrent queries with 120 nodes per query. If the `gp_enable_query_metrics` configuration parameter is off, or if the slots are exhausted, the metrics are maintained in local memory instead of in shared memory.

Value Range	Default	Set Classifications
integer `0 - 131072`	5120	master system restart

gp_interconnect_debug_retry_interval

Specifies the interval, in seconds, to log Greenplum Database interconnect debugging messages when the server configuration parameter *gp_log_interconnect* is set to `DEBUG`. The default is 10 seconds.

The log messages contain information about the interconnect communication between Greenplum Database segment instance worker processes. The information can be helpful when debugging network issues between segment instances.

Value Range	Default	Set Classifications
1 =< Integer < 4096	10	master session reload

gp_interconnect_fc_method

Specifies the flow control method used for the default Greenplum Database UDPIFC interconnect.

For capacity based flow control, senders do not send packets when receivers do not have the capacity.

Loss based flow control is based on capacity based flow control, and also tunes the sending speed according to packet losses.

Value Range	Default	Set Classifications
CAPACITY LOSS	LOSS	master session reload

gp_interconnect_proxy_addresses

Sets the proxy ports that Greenplum Database uses when the server configuration parameter *gp_interconnect_type* is set to `proxy`. Otherwise, this parameter is ignored. The default value is an empty string ("").

When the `gp_interconnect_type` parameter is set to `proxy`, You must specify a proxy port for the master, standby master, and all primary and mirror segment instances in this format:

```
<db_id>:<cont_id>:<seg_ip>:<port>[,<dbid>:<segid>:<ip>:<port> ... ]
```

For the master, standby master, and segment instance, the first three fields, *db_id*, *cont_id*, and *seg_ip* can be found in the *gp_segment_configuration* catalog table. The fourth field, *port*, is the proxy port for the Greenplum master or a segment instance.

- *db_id* is the `dbid` column in the catalog table.
- *cont_id* is the `content` column in the catalog table.
- *seg_ip* is the IP address corresponding to `address` column in the catalog table. If the `address` is a hostname, use the IP address of the hostname.
- *port* is the TCP/IP port for the segment instance proxy that you specify.

  Important: The *seg_ip* must be an IP address, not a hostname. Also, If the mapping of a segment instance hostname to the IP address changes, you must update the IP address in the parameter value.

You must specify the value as a single-quoted string. This `gpconfig` command sets the value for `gp_interconnect_proxy_addresses` as a single-quoted string. The Greenplum system consists of a master and a single segment instance.

```
gpconfig --skipvalidation -c gp_interconnect_proxy_addresses -v
 "'1:-1:192.168.180.50:35432,2:0:192.168.180.54:35000'"
```

For an example of setting `gp_interconnect_proxy_addresses`, see *Configuring Proxies for the Greenplum Interconnect*.

Value Range	Default	Set Classifications
string (maximum length - 16384 bytes)		local system restart

gp_interconnect_queue_depth

Sets the amount of data per-peer to be queued by the Greenplum Database interconnect on receivers (when data is received but no space is available to receive it the data will be dropped, and the transmitter will need to resend it) for the default UDPIFC interconnect. Increasing the depth from its default value will cause the system to use more memory, but may increase performance. It is reasonable to set this value between 1 and 10. Queries with data skew potentially perform better with an increased queue depth. Increasing this may radically increase the amount of memory used by the system.

Value Range	Default	Set Classifications
1-2048	4	master session reload

gp_interconnect_setup_timeout

Specifies the amount of time to wait for the Greenplum Database interconnect to complete setup before it times out.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	2 hours	master session reload

gp_interconnect_snd_queue_depth

Sets the amount of data per-peer to be queued by the default UDPIFC interconnect on senders. Increasing the depth from its default value will cause the system to use more memory, but may increase performance. Reasonable values for this parameter are between 1 and 4. Increasing the value might radically increase the amount of memory used by the system.

Value Range	Default	Set Classifications
1 - 4096	2	master session reload

gp_interconnect_type

Sets the networking protocol used for Greenplum Database interconnect traffic. UDPIFC specifies using UDP with flow control for interconnect traffic, and is the only value supported.

UDPIFC (the default) specifies using UDP with flow control for interconnect traffic. Specify the interconnect flow control method with *gp_interconnect_fc_method*.

With TCP as the interconnect protocol, Greenplum Database has an upper limit of 1000 segment instances - less than that if the query workload involves complex, multi-slice queries.

The PROXY value specifies using the TCP protocol, and when running queries, using a proxy for Greenplum interconnect communication between the master instance and segment instances and between two segment instances. When this parameter is set to PROXY, you must specify the proxy ports for the master and segment instances with the server configuration parameter *gp_interconnect_proxy_addresses*. For information about configuring and using proxies with the Greenplum interconnect, see *Configuring Proxies for the Greenplum Interconnect*.

Value Range	Default	Set Classifications
UDPIFC TCP PROXY	UDPIFC	local system restart

gp_log_format

Specifies the format of the server log files. If using *gp_toolkit* administrative schema, the log files must be in CSV format.

Value Range	Default	Set Classifications
csv text	csv	local system restart

gp_max_local_distributed_cache

Sets the maximum number of distributed transaction log entries to cache in the backend process memory of a segment instance.

The log entries contain information about the state of rows that are being accessed by an SQL statement. The information is used to determine which rows are visible to an SQL transaction when executing multiple simultaneous SQL statements in an MVCC environment. Caching distributed transaction log entries locally improves transaction processing speed by improving performance of the row visibility determination process.

The default value is optimal for a wide variety of SQL processing environments.

Value Range	Default	Set Classifications
integer	1024	local system restart

gp_max_packet_size

Sets the tuple-serialization chunk size for the Greenplum Database interconnect.

Value Range	Default	Set Classifications
512-65536	8192	master system restart

gp_max_plan_size

Specifies the total maximum uncompressed size of a query execution plan multiplied by the number of Motion operators (slices) in the plan. If the size of the query plan exceeds the value, the query is cancelled and an error is returned. A value of 0 means that the size of the plan is not monitored.

You can specify a value in `kB`, `MB`, or `GB`. The default unit is `kB`. For example, a value of `200` is 200kB. A value of `1GB` is the same as `1024MB` or `1048576kB`.

Value Range	Default	Set Classifications
integer	0	master superuser session

gp_max_slices

Specifies the maximum number of slices (portions of a query plan that are executed on segment instances) that can be generated by a query. If the query generates more than the specified number of slices, Greenplum Database returns an error and does not execute the query. The default value is `0`, no maximum value.

Executing a query that generates a large number of slices might affect Greenplum Database performance. For example, a query that contains `UNION` or `UNION ALL` operators over several complex views can generate a large number of slices. You can run `EXPLAIN ANALYZE` on the query to view slice statistics for the query.

Value Range	Default	Set Classifications
0 - INT_MAX	0	master session reload

gp_motion_cost_per_row

Sets the Postgres Planner cost estimate for a Motion operator to transfer a row from one segment to another, measured as a fraction of the cost of a sequential page fetch. If 0, then the value used is two times the value of *cpu_tuple_cost*.

Value Range	Default	Set Classifications
floating point	0	master session reload

gp_recursive_cte

Controls the availability of the `RECURSIVE` keyword in the `WITH` clause of a `SELECT [INTO]` command, or a `DELETE`, `INSERT` or `UPDATE` command. The keyword allows a subquery in the `WITH` clause of a command to reference itself. The default value is `false`, the `RECURSIVE` keyword is not allowed in the `WITH` clause of a command.

For information about the `RECURSIVE` keyword (Beta), see the *SELECT* command and *WITH Queries (Common Table Expressions)*.

The parameter can be set for a database system, an individual database, or a session or query.

> **Note:** This parameter was previously named `gp_recursive_cte_prototype`, but has been renamed to reflect the current status of the implementation.

Value Range	Default	Set Classifications
Boolean	true	master session restart

gp_reject_percent_threshold

For single row error handling on COPY and external table SELECTs, sets the number of rows processed before SEGMENT REJECT LIMIT *n* PERCENT starts calculating.

Value Range	Default	Set Classifications
1-*n*	300	master session reload

gp_reraise_signal

If enabled, will attempt to dump core if a fatal server error occurs.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_resgroup_memory_policy

> **Note:** The `gp_resgroup_memory_policy` server configuration parameter is enforced only when resource group-based resource management is active.

Used by a resource group to manage memory allocation to query operators.

When set to `auto`, Greenplum Database uses resource group memory limits to distribute memory across query operators, allocating a fixed size of memory to non-memory-intensive operators and the rest to memory-intensive operators.

When you specify `eager_free`, Greenplum Database distributes memory among operators more optimally by re-allocating memory released by operators that have completed their processing to operators in a later query stage.

Value Range	Default	Set Classifications
auto, eager_free	eager_free	local system superuser restart/reload

gp_resource_group_bypass

> **Note:** The `gp_resource_group_bypass` server configuration parameter is enforced only when resource group-based resource management is active.

Enables or disables the enforcement of resource group concurrent transaction limits on Greenplum Database resources. The default value is `false`, which enforces resource group transaction limits. Resource groups manage resources such as CPU, memory, and the number of concurrent transactions that are used by queries and external components such as PL/Container.

You can set this parameter to `true` to bypass resource group concurrent transaction limitations so that a query can run immediately. For example, you can set the parameter to `true` for a session to run a system catalog query or a similar query that requires a minimal amount of resources.

When you set this parameter to `true` and a run a query, the query runs in this environment:

- The query runs inside a resource group. The resource group assignment for the query does not change.
- The query memory quota is approximately 10 MB per query. The memory is allocated from resource group shared memory or global shared memory. The query fails if there is not enough shared memory available to fulfill the memory allocation request.

This parameter can be set for a session. The parameter cannot be set within a transaction or a function.

Value Range	Default	Set Classifications
Boolean	false	session

gp_resource_group_cpu_limit

Note: The `gp_resource_group_cpu_limit` server configuration parameter is enforced only when resource group-based resource management is active.

Identifies the maximum percentage of system CPU resources to allocate to resource groups on each Greenplum Database segment node.

Value Range	Default	Set Classifications
0.1 - 1.0	0.9	local system restart

gp_resource_group_memory_limit

Note: The `gp_resource_group_memory_limit` server configuration parameter is enforced only when resource group-based resource management is active.

Identifies the maximum percentage of system memory resources to allocate to resource groups on each Greenplum Database segment node.

Value Range	Default	Set Classifications
0.1 - 1.0	0.7	local system restart

Note: When resource group-based resource management is active, the memory allotted to a segment host is equally shared by active primary segments. Greenplum Database assigns memory to primary segments when the segment takes the primary role. The initial memory allotment to a primary segment does not change, even in a failover situation. This may result in a segment host utilizing more memory than the `gp_resource_group_memory_limit` setting permits.

For example, suppose your Greenplum Database cluster is utilizing the default `gp_resource_group_memory_limit` of `0.7` and a segment host named `seghost1` has 4 primary segments and 4 mirror segments. Greenplum Database assigns each primary segment on `seghost1` (`0.7 / 4 = 0.175%`) of overall system memory. If failover occurs and two mirrors on `seghost1` fail over to become primary segments, each of the original 4 primaries retain their memory allotment of `0.175`, and the two new primary segments are each allotted (`0.7 / 6 = 0.116%`) of system memory. `seghost1`'s overall memory allocation in this scenario is

```
0.7 + (0.116 * 2) = 0.932%
```

which is above the percentage configured in the `gp_resource_group_memory_limit` setting.

gp_resource_group_queuing_timeout

Note: The `gp_resource_group_queuing_timeout` server configuration parameter is enforced only when resource group-based resource management is active.

Cancel a transaction queued in a resource group that waits longer than the specified number of milliseconds. The time limit applies separately to each transaction. The default value is zero; transactions are queued indefinitely and never time out.

Value Range	Default	Set Classifications
0 - `INT_MAX` millisecs	0 millisecs	master system session reload

gp_resource_manager

Identifies the resource management scheme currently enabled in the Greenplum Database cluster. The default scheme is to use resource queues. For information about Greenplum Database resource management, see *Managing Resources*.

Value Range	Default	Set Classifications
group queue	queue	local system restart

gp_resqueue_memory_policy

> **Note:** The `gp_resqueue_memory_policy` server configuration parameter is enforced only when resource queue-based resource management is active.

Enables Greenplum memory management features. The distribution algorithm `eager_free` takes advantage of the fact that not all operators execute at the same time (in Greenplum Database 4.2 and later). The query plan is divided into stages and Greenplum Database eagerly frees memory allocated to a previous stage at the end of that stage's execution, then allocates the eagerly freed memory to the new stage.

When set to `none`, memory management is the same as in Greenplum Database releases prior to 4.1.

When set to `auto`, query memory usage is controlled by *statement_mem* and resource queue memory limits.

Value Range	Default	Set Classifications
none, auto, eager_free	eager_free	local system restart/reload

gp_resqueue_priority

> **Note:** The `gp_resqueue_priority` server configuration parameter is enforced only when resource queue-based resource management is active.

Enables or disables query prioritization. When this parameter is disabled, existing priority settings are not evaluated at query run time.

Value Range	Default	Set Classifications
Boolean	on	local system restart

gp_resqueue_priority_cpucores_per_segment

Note: The `gp_resqueue_priority_cpucores_per_segment` server configuration parameter is enforced only when resource queue-based resource management is active.

Specifies the number of CPU units allocated per segment instance. For example, if a Greenplum Database cluster has 10-core segment hosts that are configured with four segments, set the value for the segment instances to 2.5. For the master instance, the value would be 10. A master host typically has only the master instance running on it, so the value for the master should reflect the usage of all available CPU cores.

Incorrect settings can result in CPU under-utilization or query prioritization not working as designed.

Value Range	Default	Set Classifications
0.1 - 512.0	4	local system restart

gp_resqueue_priority_sweeper_interval

Note: The `gp_resqueue_priority_sweeper_interval` server configuration parameter is enforced only when resource queue-based resource management is active.

Specifies the interval at which the sweeper process evaluates current CPU usage. When a new statement becomes active, its priority is evaluated and its CPU share determined when the next interval is reached.

Value Range	Default	Set Classifications
500 - 15000 ms	1000	local system restart

gp_role

The role of this server process '' set to *dispatch* for the master and *execute* for a segment.

Value Range	Default	Set Classifications
dispatch execute utility		read only

gp_safefswritesize

Specifies a minimum size for safe write operations to append-optimized tables in a non-mature file system. When a number of bytes greater than zero is specified, the append-optimized writer adds padding data up to that number in order to prevent data corruption due to file system errors. Each non-mature file system has a known safe write size that must be specified here when using Greenplum Database with that type of file system. This is commonly set to a multiple of the extent size of the file system; for example, Linux ext3 is 4096 bytes, so a value of 32768 is commonly used.

Value Range	Default	Set Classifications
integer	0	local system restart

gp_segment_connect_timeout

Time that the Greenplum interconnect will try to connect to a segment instance over the network before timing out. Controls the network connection timeout between master and primary segments, and primary to mirror segment replication processes.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	10min	local system reload

gp_segments_for_planner

Sets the number of primary segment instances for the Postgres Planner to assume in its cost and size estimates. If 0, then the value used is the actual number of primary segments. This variable affects the Postgres Planner's estimates of the number of rows handled by each sending and receiving process in Motion operators.

Value Range	Default	Set Classifications
0-*n*	0	master session reload

gp_server_version

Reports the version number of the server as a string. A version modifier argument might be appended to the numeric portion of the version string, example: *5.0.0 beta*.

Value Range	Default	Set Classifications
String. Examples: *5.0.0*	n/a	read only

gp_server_version_num

Reports the version number of the server as an integer. The number is guaranteed to always be increasing for each version and can be used for numeric comparisons. The major version is represented as is, the minor and patch versions are zero-padded to always be double digit wide.

Value Range	Default	Set Classifications
Mmmpp where *M* is the major version, *mm* is the minor version zero-padded and *pp* is the patch version zero-padded. Example: 50000	n/a	read only

gp_session_id

A system assigned ID number for a client session. Starts counting from 1 when the master instance is first started.

Value Range	Default	Set Classifications
1-*n*	14	read only

gp_set_proc_affinity

If enabled, when a Greenplum server process (postmaster) is started it will bind to a CPU.

Value Range	Default	Set Classifications
Boolean	off	master system restart

gp_set_read_only

Set to on to disable writes to the database. Any in progress transactions must finish before read-only mode takes affect.

Value Range	Default	Set Classifications
Boolean	off	master system restart

gp_statistics_pullup_from_child_partition

Enables the use of statistics from child tables when planning queries on the parent table by the Postgres Planner.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_statistics_use_fkeys

When enabled, the Postgres Planner will use the statistics of the referenced column in the parent table when a column is foreign key reference to another table instead of the statistics of the column itself.

Note: This parameter is deprecated and will be removed in a future Greenplum Database release.

Value Range	Default	Set Classifications
Boolean	off	master session reload

gp_use_legacy_hashops

For a table that is defined with a `DISTRIBUTED BY` *`key_column`* clause, this parameter controls the hash algorithm that is used to distribute table data among segment instances. The default value is `false`, use the jump consistent hash algorithm.

Setting the value to `true` uses the modulo hash algorithm that is compatible with Greenplum Database 5.x and earlier releases.

Value Range	Default	Set Classifications
Boolean	false	master session reload

gp_vmem_idle_resource_timeout

If a database session is idle for longer than the time specified, the session will free system resources (such as shared memory), but remain connected to the database. This allows more concurrent connections to the database at one time.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	18s	master system reload

gp_vmem_protect_limit

> **Note:** The `gp_vmem_protect_limit` server configuration parameter is enforced only when resource queue-based resource management is active.

Sets the amount of memory (in number of MBs) that all postgres processes of an active segment instance can consume. If a query causes this limit to be exceeded, memory will not be allocated and the query will fail. Note that this is a local parameter and must be set for every segment in the system (primary and mirrors). When setting the parameter value, specify only the numeric value. For example, to specify 4096MB, use the value `4096`. Do not add the units `MB` to the value.

To prevent over-allocation of memory, these calculations can estimate a safe `gp_vmem_protect_limit` value.

First calculate the value *gp_vmem*. This is the Greenplum Database memory available on a host

```
gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM)) / 1.7
```

where *SWAP* is the host swap space and *RAM* is the RAM on the host in GB.

Next, calculate the *max_acting_primary_segments*. This is the maximum number of primary segments that can be running on a host when mirror segments are activated due to a failure. With mirrors arranged in a 4-host block with 8 primary segments per host, for example, a single segment host failure would activate two or three mirror segments on each remaining host in the failed host's block. The *max_acting_primary_segments* value for this configuration is 11 (8 primary segments plus 3 mirrors activated on failure).

This is the calculation for `gp_vmem_protect_limit`. The value should be converted to MB.

```
gp_vmem_protect_limit = gp_vmem / acting_primary_segments
```

For scenarios where a large number of workfiles are generated, this is the calculation for *gp_vmem* that accounts for the workfiles.

```
gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM - (300KB
 * total_#_workfiles))) / 1.7
```

For information about monitoring and managing workfile usage, see the *Greenplum Database Administrator Guide*.

Based on the *gp_vmem* value you can calculate the value for the `vm.overcommit_ratio` operating system kernel parameter. This parameter is set when you configure each Greenplum Database host.

```
vm.overcommit_ratio = (RAM - (0.026 * gp_vmem)) / RAM
```

> **Note:** The default value for the kernel parameter `vm.overcommit_ratio` in Red Hat Enterprise Linux is 50.

For information about the kernel parameter, see the *Greenplum Database Installation Guide*.

Value Range	Default	Set Classifications
integer	8192	local system restart

gp_vmem_protect_segworker_cache_limit

If a query executor process consumes more than this configured amount, then the process will not be cached for use in subsequent queries after the process completes. Systems with lots of connections or idle processes may want to reduce this number to free more memory on the segments. Note that this is a local parameter and must be set for every segment.

Value Range	Default	Set Classifications
number of megabytes	500	local system restart

gp_workfile_compression

Specifies whether the temporary files created, when a hash aggregation or hash join operation spills to disk, are compressed.

If your Greenplum Database installation uses serial ATA (SATA) disk drives, enabling compression might help to avoid overloading the disk subsystem with IO operations.

Value Range	Default	Set Classifications
Boolean	off	master session reload

gp_workfile_limit_files_per_query

Sets the maximum number of temporary spill files (also known as workfiles) allowed per query per segment. Spill files are created when executing a query that requires more memory than it is allocated. The current query is terminated when the limit is exceeded.

Set the value to 0 (zero) to allow an unlimited number of spill files. master session reload

Value Range	Default	Set Classifications
integer	100000	master session reload

gp_workfile_limit_per_query

Sets the maximum disk size an individual query is allowed to use for creating temporary spill files at each segment. The default value is 0, which means a limit is not enforced.

Value Range	Default	Set Classifications
kilobytes	0	master session reload

gp_workfile_limit_per_segment

Sets the maximum total disk size that all running queries are allowed to use for creating temporary spill files at each segment. The default value is 0, which means a limit is not enforced.

Value Range	Default	Set Classifications
kilobytes	0	local system restart

gpperfmon_port

Sets the port on which all data collection agents communicate with the master.

Value Range	Default	Set Classifications
integer	8888	master system restart

ignore_checksum_failure

Only has effect if *data_checksums* is enabled.

Greenplum Database uses checksums to prevent loading data that has been corrupted in the file system into memory managed by database processes.

By default, when a checksum verify error occurs when reading a heap data page, Greenplum Database generates an error and prevents the page from being loaded into managed memory. When `ignore_checksum_failure` is set to on and a checksum verify failure occurs, Greenplum Database generates a warning, and allows the page to be read into managed memory. If the page is then updated it is saved to disk and replicated to the mirror. If the page header is corrupt an error is reported even if this option is enabled.

> **Warning:** Setting `ignore_checksum_failure` to on may propagate or hide data corruption or lead to other serious problems. However, if a checksum failure has already been detected and the page header is uncorrupted, setting `ignore_checksum_failure` to on may allow you to bypass the error and recover undamaged tuples that may still be present in the table.

The default setting is off, and it can only be changed by a superuser.

Value Range	Default	Set Classifications
Boolean	off	local system restart

integer_datetimes

Reports whether PostgreSQL was built with support for 64-bit-integer dates and times.

Value Range	Default	Set Classifications
Boolean	on	read only

IntervalStyle

Sets the display format for interval values. The value *sql_standard* produces output matching SQL standard interval literals. The value *postgres* produces output matching PostgreSQL releases prior to 8.4 when the *DateStyle* parameter was set to ISO.

The value *postgres_verbose* produces output matching Greenplum releases prior to 3.3 when the *DateStyle* parameter was set to non-ISO output.

The value *iso_8601* will produce output matching the time interval *format with designators* defined in section 4.4.3.2 of ISO 8601. See the *PostgreSQL 9.4 documentation* for more information.

Value Range	Default	Set Classifications
postgres postgres_verbose sql_standard iso_8601	postgres	master session reload

join_collapse_limit

The Postgres Planner will rewrite explicit inner `JOIN` constructs into lists of `FROM` items whenever a list of no more than this many items in total would result. By default, this variable is set the same as *from_collapse_limit*, which is appropriate for most uses. Setting it to 1 prevents any reordering of inner JOINs. Setting this variable to a value between 1 and *from_collapse_limit* might be useful to trade off planning time against the quality of the chosen plan (higher values produce better plans).

Value Range	Default	Set Classifications
1-*n*	20	master session reload

krb_caseins_users

Sets whether Kerberos user names should be treated case-insensitively. The default is case sensitive (off).

Value Range	Default	Set Classifications
Boolean	off	master system reload

krb_server_keyfile

Sets the location of the Kerberos server key file.

Value Range	Default	Set Classifications
path and file name	unset	master system restart

lc_collate

Reports the locale in which sorting of textual data is done. The value is determined when the Greenplum Database array is initialized.

Value Range	Default	Set Classifications
<system dependent>		read only

lc_ctype

Reports the locale that determines character classifications. The value is determined when the Greenplum Database array is initialized.

Value Range	Default	Set Classifications
<system dependent>		read only

lc_messages

Sets the language in which messages are displayed. The locales available depends on what was installed with your operating system - use *locale -a* to list available locales. The default value is inherited from the execution environment of the server. On some systems, this locale category does not exist. Setting this variable will still work, but there will be no effect. Also, there is a chance that no translated messages for the desired language exist. In that case you will continue to see the English messages.

Value Range	Default	Set Classifications
<system dependent>		local system restart

lc_monetary

Sets the locale to use for formatting monetary amounts, for example with the *to_char* family of functions. The locales available depends on what was installed with your operating system - use *locale -a* to list available locales. The default value is inherited from the execution environment of the server.

Value Range	Default	Set Classifications
<system dependent>		local system restart

lc_numeric

Sets the locale to use for formatting numbers, for example with the *to_char* family of functions. The locales available depends on what was installed with your operating system - use *locale -a* to list available locales. The default value is inherited from the execution environment of the server.

Value Range	Default	Set Classifications
<system dependent>		local system restart

lc_time

This parameter currently does nothing, but may in the future.

Value Range	Default	Set Classifications
<system dependent>		local system restart

listen_addresses

Specifies the TCP/IP address(es) on which the server is to listen for connections from client applications - a comma-separated list of host names and/or numeric IP addresses. The special entry * corresponds to all available IP interfaces. If the list is empty, only UNIX-domain sockets can connect.

Value Range	Default	Set Classifications
localhost, host names, IP addresses, * (all available IP interfaces)	*	master system restart

local_preload_libraries

Comma separated list of shared library files to preload at the start of a client session.

Value Range	Default	Set Classifications
		local system restart

lock_timeout

Abort any statement that waits longer than the specified number of milliseconds while attempting to acquire a lock on a table, index, row, or other database object. The time limit applies separately to each lock acquisition attempt. The limit applies both to explicit locking requests (such as `LOCK TABLE` or `SELECT FOR UPDATE`) and to implicitly-acquired locks. If `log_min_error_statement` is set to `ERROR` or lower, Greenplum Database logs the statement that timed out. A value of zero (the default) turns off this lock wait monitoring.

Unlike *`statement_timeout`*, this timeout can only occur while waiting for locks. Note that if `statement_timeout` is nonzero, it is rather pointless to set `lock_timeout` to the same or larger value, since the statement timeout would always trigger first.

Greenplum Database uses the *`deadlock_timeout`* and *`gp_global_deadlock_detector_period`* to trigger local and global deadlock detection. Note that if `lock_timeout` is turned on and set to a value smaller than these deadlock detection timeouts, Greenplum Database will abort a statement before it would ever trigger a deadlock check in that session.

> **Note:** Setting `lock_timeout` in `postgresql.conf` is not recommended because it would affect all sessions

Value Range	Default	Set Classifications
0 - `INT_MAX` millisecs	0 millisecs	master session reload

log_autostats

Logs information about automatic `ANALYZE` operations related to *gp_autostats_mode* and *gp_autostats_on_change_threshold*.

Value Range	Default	Set Classifications
Boolean	off	master session reload superuser

log_connections

This outputs a line to the server log detailing each successful connection. Some client programs, like psql, attempt to connect twice while determining if a password is required, so duplicate "connection received" messages do not always indicate a problem.

Value Range	Default	Set Classifications
Boolean	off	local system restart

log_disconnections

This outputs a line in the server log at termination of a client session, and includes the duration of the session.

Value Range	Default	Set Classifications
Boolean	off	local system restart

log_dispatch_stats

When set to "on," this parameter adds a log message with verbose information about the dispatch of the statement.

Value Range	Default	Set Classifications
Boolean	off	local system restart

log_duration

Causes the duration of every completed statement which satisfies *log_statement* to be logged.

Value Range	Default	Set Classifications
Boolean	off	master session reload superuser

log_error_verbosity

Controls the amount of detail written in the server log for each message that is logged.

Value Range	Default	Set Classifications
TERSE DEFAULT VERBOSE	DEFAULT	master session reload superuser

log_executor_stats

For each query, write performance statistics of the query executor to the server log. This is a crude profiling instrument. Cannot be enabled together with *log_statement_stats*.

Value Range	Default	Set Classifications
Boolean	off	local system restart

log_hostname

By default, connection log messages only show the IP address of the connecting host. Turning on this option causes logging of the IP address and host name of the Greenplum Database master. Note that depending on your host name resolution setup this might impose a non-negligible performance penalty.

Value Range	Default	Set Classifications
Boolean	off	master system restart

log_min_duration_statement

Logs the statement and its duration on a single log line if its duration is greater than or equal to the specified number of milliseconds. Setting this to 0 will print all statements and their durations. -1 disables the feature. For example, if you set it to 250 then all SQL statements that run 250ms or longer will be logged. Enabling this option can be useful in tracking down unoptimized queries in your applications.

Value Range	Default	Set Classifications
number of milliseconds, 0, -1	-1	master session reload superuser

log_min_error_statement

Controls whether or not the SQL statement that causes an error condition will also be recorded in the server log. All SQL statements that cause an error of the specified level or higher are logged. The default is ERROR. To effectively turn off logging of failing statements, set this parameter to PANIC.

Value Range	Default	Set Classifications
DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 INFO NOTICE WARNING ERROR FATAL PANIC	ERROR	master session reload superuser

log_min_messages

Controls which message levels are written to the server log. Each level includes all the levels that follow it. The later the level, the fewer messages are sent to the log.

If the Greenplum Database PL/Container extension is installed. This parameter also controls the PL/Container log level. For information about the extension, see *../../analytics/pl_container.xml*.

Value Range	Default	Set Classifications
DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 INFO NOTICE WARNING LOG ERROR FATAL PANIC	WARNING	master session reload superuser

log_parser_stats

For each query, write performance statistics of the query parser to the server log. This is a crude profiling instrument. Cannot be enabled together with *log_statement_stats*.

Value Range	Default	Set Classifications
Boolean	off	master session reload superuser

log_planner_stats

For each query, write performance statistics of the Postgres Planner to the server log. This is a crude profiling instrument. Cannot be enabled together with *log_statement_stats*.

Value Range	Default	Set Classifications
Boolean	off	master session reload superuser

log_rotation_age

Determines the amount of time Greenplum Database writes messages to the active log file. When this amount of time has elapsed, the file is closed and a new log file is created. Set to zero to disable time-based creation of new log files.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	1d	local system restart

log_rotation_size

Determines the size of an individual log file that triggers rotation. When the log file size is equal to or greater than this size, the file is closed and a new log file is created. Set to zero to disable size-based creation of new log files.

The maximum value is INT_MAX/1024. If an invalid value is specified, the default value is used. INT_MAX is the largest value that can be stored as an integer on your system.

Value Range	Default	Set Classifications
number of kilobytes	1048576	local system restart

log_statement

Controls which SQL statements are logged. DDL logs all data definition commands like CREATE, ALTER, and DROP commands. MOD logs all DDL statements, plus INSERT, UPDATE, DELETE, TRUNCATE, and COPY FROM. PREPARE and EXPLAIN ANALYZE statements are also logged if their contained command is of an appropriate type.

Value Range	Default	Set Classifications
NONE DDL MOD ALL	ALL	master session reload superuser

log_statement_stats

For each query, write total performance statistics of the query parser, planner, and executor to the server log. This is a crude profiling instrument.

Value Range	Default	Set Classifications
Boolean	off	master session reload superuser

log_temp_files

Controls logging of temporary file names and sizes. Temporary files can be created for sorts, hashes, temporary query results and spill files. A log entry is made in `pg_log` for each temporary file when it is deleted. Depending on the source of the temporary files, the log entry could be created on either the

master and/or segments. A `log_temp_files` value of zero logs all temporary file information, while positive values log only files whose size is greater than or equal to the specified number of kilobytes. The default setting is `-1`, which disables logging. Only superusers can change this setting.

Value Range	Default	Set Classifications
Integer	-1	local system restart

log_timezone

Sets the time zone used for timestamps written in the log. Unlike *TimeZone*, this value is system-wide, so that all sessions will report timestamps consistently. The default is `unknown`, which means to use whatever the system environment specifies as the time zone.

Value Range	Default	Set Classifications
string	unknown	local system restart

log_truncate_on_rotation

Truncates (overwrites), rather than appends to, any existing log file of the same name. Truncation will occur only when a new file is being opened due to time-based rotation. For example, using this setting in combination with a log_filename such as `gpseg#-%H.log` would result in generating twenty-four hourly log files and then cyclically overwriting them. When off, pre-existing files will be appended to in all cases.

Value Range	Default	Set Classifications
Boolean	off	local system restart

maintenance_work_mem

Specifies the maximum amount of memory to be used in maintenance operations, such as `VACUUM` and `CREATE INDEX`. It defaults to 16 megabytes (16MB). Larger settings might improve performance for vacuuming and for restoring database dumps.

Value Range	Default	Set Classifications
Integer	16	local system restart

max_appendonly_tables

Sets the maximum number of concurrent transactions that can write to or update append-optimized tables. Transactions that exceed the maximum return an error.

Operations that are counted are `INSERT`, `UPDATE`, `COPY`, and `VACUUM` operations. The limit is only for in-progress transactions. Once a transaction ends (either aborted or committed), it is no longer counted against this limit.

For operations against a partitioned table, each subpartition (child table) that is an append-optimized table and is changed counts as a single table towards the maximum. For example, a partitioned table `p_tbl` is defined with three subpartitions that are append-optimized tables `p_tbl_ao1`, `p_tbl_ao2`, and `p_tbl_ao3`. An `INSERT` or `UPDATE` command against the partitioned table `p_tbl` that changes append-optimized tables `p_tbl_ao1` and `p_tbl_ao2` is counted as two transactions.

Increasing the limit allocates more shared memory on the master host at server start.

Value Range	Default	Set Classifications
integer > 0	10000	master system restart

max_connections

The maximum number of concurrent connections to the database server. In a Greenplum Database system, user client connections go through the Greenplum master instance only. Segment instances should allow 5-10 times the amount as the master. When you increase this parameter, *max_prepared_transactions* must be increased as well. For more information about limiting concurrent connections, see "Configuring Client Authentication" in the *Greenplum Database Administrator Guide*.

Increasing this parameter may cause Greenplum Database to request more shared memory. Increasing this parameter might cause Greenplum Database to request more shared memory. See *shared_buffers* for information about Greenplum server instance shared memory buffers.

Value Range	Default	Set Classifications
10 - 8388607	250 on master 750 on segments	local system restart

max_files_per_process

Sets the maximum number of simultaneously open files allowed to each server subprocess. If the kernel is enforcing a safe per-process limit, you don't need to worry about this setting. Some platforms such as BSD, the kernel will allow individual processes to open many more files than the system can really support.

Value Range	Default	Set Classifications
integer	1000	local system restart

max_function_args

Reports the maximum number of function arguments.

Value Range	Default	Set Classifications
integer	100	read only

max_identifier_length

Reports the maximum identifier length.

Value Range	Default	Set Classifications
integer	63	read only

max_index_keys

Reports the maximum number of index keys.

Value Range	Default	Set Classifications
integer	32	read only

max_locks_per_transaction

The shared lock table is created with room to describe locks on *max_locks_per_transaction* * (*max_connections* + *max_prepared_transactions*) objects, so no more than this many distinct objects can be locked at any one time. This is not a hard limit on the number of locks taken by any one transaction, but rather a maximum average value. You might need to raise this value if you have clients that touch many different tables in a single transaction.

Value Range	Default	Set Classifications
integer	128	local system restart

max_prepared_transactions

Sets the maximum number of transactions that can be in the prepared state simultaneously. Greenplum uses prepared transactions internally to ensure data integrity across the segments. This value must be at least as large as the value of *max_connections* on the master. Segment instances should be set to the same value as the master.

Value Range	Default	Set Classifications
integer	250 on master 250 on segments	local system restart

max_resource_portals_per_transaction

> **Note:** The `max_resource_portals_per_transaction` server configuration parameter is enforced only when resource queue-based resource management is active.

Sets the maximum number of simultaneously open user-declared cursors allowed per transaction. Note that an open cursor will hold an active query slot in a resource queue. Used for resource management.

Value Range	Default	Set Classifications
integer	64	master system restart

max_resource_queues

Note: The `max_resource_queues` server configuration parameter is enforced only when resource queue-based resource management is active.

Sets the maximum number of resource queues that can be created in a Greenplum Database system. Note that resource queues are system-wide (as are roles) so they apply to all databases in the system.

Value Range	Default	Set Classifications
integer	9	master system restart

max_slot_wal_keep_size

Sets the maximum size in megabytes of Write-Ahead Logging (WAL) files on disk per segment instance that can be reserved when Greenplum streams data to the mirror segment instance or standby master to keep it synchronized with the corresponding primary segment instance or master. The default is -1, Greenplum can retain an unlimited amount of WAL files on disk.

If the file size exceeds the maximum size, the files are released and are available for deletion. A mirror or standby may no longer be able to continue replication due to removal of required WAL files.

Value Range	Default	Set Classifications
Integer	-1	local system restart

max_stack_depth

Specifies the maximum safe depth of the server's execution stack. The ideal setting for this parameter is the actual stack size limit enforced by the kernel (as set by *ulimit -s* or local equivalent), less a safety margin of a megabyte or so. Setting the parameter higher than the actual kernel limit will mean that a runaway recursive function can crash an individual backend process.

Value Range	Default	Set Classifications
number of kilobytes	2MB	local system restart

max_statement_mem

Sets the maximum memory limit for a query. Helps avoid out-of-memory errors on a segment host during query processing as a result of setting *statement_mem* too high.

Taking into account the configuration of a single segment host, calculate `max_statement_mem` as follows:

```
(seghost_physical_memory) / (average_number_concurrent_queries)
```

When changing both `max_statement_mem` and `statement_mem`, `max_statement_mem` must be changed first, or listed first in the `postgresql.conf` file.

Value Range	Default	Set Classifications
number of kilobytes	2000MB	master session reload superuser

memory_spill_ratio

> **Note:** The `memory_spill_ratio` server configuration parameter is enforced only when resource group-based resource management is active.

Sets the memory usage threshold percentage for memory-intensive operators in a transaction. When a transaction reaches this threshold, it spills to disk.

The default `memory_spill_ratio` percentage is the value defined for the resource group assigned to the currently active role. You can set `memory_spill_ratio` at the session level to selectively set this limit on a per-query basis. For example, if you have a specific query that spills to disk and requires more memory, you may choose to set a larger `memory_spill_ratio` to increase the initial memory allocation.

You can specify an integer percentage value from 0 to 100 inclusive. If you specify a value of 0, Greenplum Database uses the *`statement_mem`* server configuration parameter value to control the initial query operator memory amount.

Value Range	Default	Set Classifications
0 - 100	20	master session reload

optimizer

Enables or disables GPORCA when running SQL queries. The default is `on`. If you disable GPORCA, Greenplum Database uses only the Postgres Planner.

GPORCA co-exists with the Postgres Planner. With GPORCA enabled, Greenplum Database uses GPORCA to generate an execution plan for a query when possible. If GPORCA cannot be used, then the Postgres Planner is used.

The `optimizer` parameter can be set for a database system, an individual database, or a session or query.

For information about the Postgres Planner and GPORCA, see *Querying Data* in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
Boolean	on	master session reload

optimizer_analyze_root_partition

For a partitioned table, controls whether the `ROOTPARTITION` keyword is required to collect root partition statistics when the `ANALYZE` command is run on the table. GPORCA uses the root partition statistics when generating a query plan. The Postgres Planner does not use these statistics.

The default setting for the parameter is `on`, the `ANALYZE` command can collect root partition statistics without the `ROOTPARTITION` keyword. Root partition statistics are collected when you run `ANALYZE` on the root partition, or when you run `ANALYZE` on a child leaf partition of the partitioned table and the other child leaf partitions have statistics. When the value is `off`, you must run `ANALZYE ROOTPARTITION` to collect root partition statistics.

When the value of the server configuration parameter *`optimizer`* is `on` (the default), the value of this parameter should also be `on`. For information about collecting table statistics on partitioned tables, see *`ANALYZE`*.

For information about the Postgres Planner and GPORCA, see *Querying Data* in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
Boolean	on	master session reload

optimizer_array_expansion_threshold

When GPORCA is enabled (the default) and is processing a query that contains a predicate with a constant array, the `optimizer_array_expansion_threshold` parameter limits the optimization process based on the number of constants in the array. If the array in the query predicate contains more than the number elements specified by parameter, GPORCA disables the transformation of the predicate into its disjunctive normal form during query optimization.

The default value is 100.

For example, when GPORCA is executing a query that contains an `IN` clause with more than 100 elements, GPORCA does not transform the predicate into its disjunctive normal form during query optimization to reduce optimization time consume less memory. The difference in query processing can be seen in the filter condition for the `IN` clause of the query `EXPLAIN` plan.

Changing the value of this parameter changes the trade-off between a shorter optimization time and lower memory consumption, and the potential benefits from constraint derivation during query optimization, for example conflict detection and partition elimination.

The parameter can be set for a database system, an individual database, or a session or query.

Value Range	Default	Set Classifications
Integer > 0	25	master session reload

optimizer_control

Controls whether the server configuration parameter `optimizer` can be changed with `SET`, the `RESET` command, or the Greenplum Database utility `gpconfig`. If the `optimizer_control` parameter value is `on`, users can set the `optimizer` parameter. If the `optimizer_control` parameter value is `off`, the `optimizer` parameter cannot be changed.

Value Range	Default	Set Classifications
Boolean	on	master system restart superuser

optimizer_cte_inlining_bound

When GPORCA is enabled (the default), this parameter controls the amount of inlining performed for common table expression (CTE) queries (queries that contain a `WHERE` clause). The default value, 0, disables inlining.

The parameter can be set for a database system, an individual database, or a session or query.

Value Range	Default	Set Classifications
Decimal >= 0	0	master session reload

optimizer_enable_associativity

When GPORCA is enabled (the default), this parameter controls whether the join associativity transform is enabled during query optimization. The transform analyzes join orders. For the default value `off`, only the GPORCA dynamic programming algorithm for analyzing join orders is enabled. The join associativity transform largely duplicates the functionality of the newer dynamic programming algorithm.

If the value is `on`, GPORCA can use the associativity transform during query optimization.

The parameter can be set for a database system, an individual database, or a session or query.

For information about GPORCA, see *About GPORCA* in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
Boolean	off	master session reload

optimizer_enable_dml

When GPORCA is enabled (the default) and this parameter is `true` (the default), GPORCA attempts to execute DML commands such as `INSERT`, `UPDATE`, and `DELETE`. If GPORCA cannot execute the command, Greenplum Database falls back to the Postgres Planner.

When set to `false`, Greenplum Database always falls back to the Postgres Planner when performing DML commands.

The parameter can be set for a database system, an individual database, or a session or query.

For information about GPORCA, see *About GPORCA* in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
Boolean	true	master session reload

optimizer_enable_master_only_queries

When GPORCA is enabled (the default), this parameter allows GPORCA to execute catalog queries that run only on the Greenplum Database master. For the default value `off`, only the Postgres Planner can execute catalog queries that run only on the Greenplum Database master.

The parameter can be set for a database system, an individual database, or a session or query.

> **Note:** Enabling this parameter decreases performance of short running catalog queries. To avoid this issue, set this parameter only for a session or a query.

For information about GPORCA, see *About GPORCA* in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
Boolean	off	master session reload

optimizer_force_agg_skew_avoidance

When GPORCA is enabled (the default), this parameter affects the query plan alternatives that GPORCA considers when 3 stage aggregate plans are generated. When the value is `true`, the default, GPORCA considers only 3 stage aggregate plans where the intermediate aggregation uses the `GROUP BY` and `DISTINCT` columns for distribution to reduce the effects of processing skew.

If the value is `false`, GPORCA can also consider a plan that uses `GROUP BY` columns for distribution. These plans might perform poorly when processing skew is present.

For information about GPORCA, see *About GPORCA* in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
Boolean	true	master session reload

optimizer_force_multistage_agg

For the default settings, GPORCA is enabled and this parameter is `false`, GPORCA makes a cost-based choice between a one- or two-stage aggregate plan for a scalar distinct qualified aggregate. When `true`, GPORCA chooses a multi-stage aggregate plan when such a plan alternative is generated.

The parameter can be set for a database system, an individual database, or a session or query.

Value Range	Default	Set Classifications
Boolean	true	master session reload

optimizer_force_three_stage_scalar_dqa

For the default settings, GPORCA is enabled and this parameter is `true`, GPORCA chooses a plan with multistage aggregates when such a plan alternative is generated. When the value is `false`, GPORCA makes a cost based choice rather than a heuristic choice.

The parameter can be set for a database system, an individual database, or a session, or query.

Value Range	Default	Set Classifications
Boolean	true	master session reload

optimizer_join_arity_for_associativity_commutativity

The value is an optimization hint to limit the number of join associativity and join commutativity transformations explored during query optimization. The limit controls the alternative plans that GPORCA considers during query optimization. For example, the default value of 18 is an optimization hint for GPORCA to stop exploring join associativity and join commutativity transformations when an n-ary join operator has more than 18 children during optimization.

For a query with a large number of joins, specifying a lower value improves query performance by limiting the number of alternate query plans that GPORCA evaluates. However, setting the value too low might cause GPORCA to generate a query plan that performs sub-optimally.

This parameter has no effect when the `optimizer_join_order` parameter is set to `query` or `greedy`.

This parameter can be set for a database system or a session.

Value Range	Default	Set Classifications
integer > 0	18	local system reload

optimizer_join_order

When GPORCA is enabled, this parameter sets the optimization level for join ordering during query optimization by specifying which types of join ordering alternatives to evaluate.

- `query` - Uses the join order specified in the query.
- `greedy` - Evaluates the join order specified in the query and alternatives based on minimum cardinalities of the relations in the joins.
- `exhaustive` - Applies transformation rules to find and evaluate all join ordering alternatives.

The default value is `exhaustive`. Setting this parameter to `query` or `greedy` can generate a suboptimal query plan. However, if the administrator is confident that a satisfactory plan is generated with the `query` or `greedy` setting, query optimization time can be improved by setting the parameter to the lower optimization level.

Setting this parameter to `query` or `greedy` overrides the `optimizer_join_order_threshold` and `optimizer_join_arity_for_associativity_commutativity` parameters.

This parameter can be set for an individual database, a session, or a query.

Value Range	Default	Set Classifications
query greedy exhaustive	exhaustive	master session reload

optimizer_join_order_threshold

When GPORCA is enabled (the default), this parameter sets the maximum number of join children for which GPORCA will use the dynamic programming-based join ordering algorithm. You can set this value for a single query or for an entire session.

This parameter has no effect when the `optimizer_join_query` parameter is set to `query` or `greedy`.

Value Range	Default	Set Classifications
0 - 12	10	master session reload

optimizer_mdcache_size

Sets the maximum amount of memory on the Greenplum Database master that GPORCA uses to cache query metadata (optimization data) during query optimization. The memory limit session based. GPORCA caches query metadata during query optimization with the default settings: GPORCA is enabled and *`optimizer_metadata_caching`* is on.

The default value is 16384 (16MB). This is an optimal value that has been determined through performance analysis.

You can specify a value in KB, MB, or GB. The default unit is KB. For example, a value of 16384 is 16384KB. A value of 1GB is the same as 1024MB or 1048576KB. If the value is 0, the size of the cache is not limited.

This parameter can be set for a database system, an individual database, or a session or query.

Value Range	Default	Set Classifications
Integer >= 0	16384	master session reload

optimizer_metadata_caching

When GPORCA is enabled (the default), this parameter specifies whether GPORCA caches query metadata (optimization data) in memory on the Greenplum Database master during query optimization. The default for this parameter is `on`, enable caching. The cache is session based. When a session ends, the cache is released. If the amount of query metadata exceeds the cache size, then old, unused metadata is evicted from the cache.

If the value is `off`, GPORCA does not cache metadata during query optimization.

This parameter can be set for a database system, an individual database, or a session or query.

The server configuration parameter *optimizer_mdcache_size* controls the size of the query metadata cache.

Value Range	Default	Set Classifications
Boolean	on	master session reload

optimizer_minidump

GPORCA generates minidump files to describe the optimization context for a given query. The information in the file is not in a format that can be easily used for debugging or troubleshooting. The minidump file is located under the master data directory and uses the following naming format:

`Minidump_`*`date_time`*`.mdp`

The minidump file contains this query related information:

- Catalog objects including data types, tables, operators, and statistics required by GPORCA
- An internal representation (DXL) of the query
- An internal representation (DXL) of the plan produced by GPORCA
- System configuration information passed to GPORCA such as server configuration parameters, cost and statistics configuration, and number of segments
- A stack trace of errors generated while optimizing the query

Setting this parameter to `ALWAYS` generates a minidump for all queries. Set this parameter to `ONERROR` to minimize total optimization time.

For information about GPORCA, see *About GPORCA* in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
ONERROR ALWAYS	ONERROR	master session reload

optimizer_nestloop_factor

This parameter adds a costing factor to GPORCA to prioritize hash joins instead of nested loop joins during query optimization. The default value of 1024 was chosen after evaluating numerous workloads with uniformly distributed data. 1024 should be treated as the practical upper bound setting for this parameter. If you find the GPORCA selects hash joins more often than it should, reduce the value to shift the costing factor in favor of nested loop joins.

The parameter can be set for a database system, an individual database, or a session or query.

Value Range	Default	Set Classifications
INT_MAX > 1	1024	master session reload

optimizer_parallel_union

When GPORCA is enabled (the default), `optimizer_parallel_union` controls the amount of parallelization that occurs for queries that contain a `UNION` or `UNION ALL` clause.

When the value is `off`, the default, GPORCA generates a query plan where each child of an APPEND(UNION) operator is in the same slice as the APPEND operator. During query execution, the children are executed in a sequential manner.

When the value is `on`, GPORCA generates a query plan where a redistribution motion node is under an APPEND(UNION) operator. During query execution, the children and the parent APPEND operator are on different slices, allowing the children of the APPEND(UNION) operator to execute in parallel on segment instances.

The parameter can be set for a database system, an individual database, or a session or query.

Value Range	Default	Set Classifications
boolean	off	master session reload

optimizer_penalize_skew

When GPORCA is enabled (the default), this parameter allows GPORCA to penalize the local cost of a HashJoin with a skewed Redistribute Motion as child to favor a Broadcast Motion during query optimization. The default value is `true`.

GPORCA determines there is skew for a Redistribute Motion when the NDV (number of distinct values) is less than the number of segments.

The parameter can be set for a database system, an individual database, or a session or query.

For information about GPORCA, see *About GPORCA* in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
Boolean	true	master session reload

optimizer_print_missing_stats

When GPORCA is enabled (the default), this parameter controls the display of table column information about columns with missing statistics for a query. The default value is `true`, display the column information to the client. When the value is `false`, the information is not sent to the client.

The information is displayed during query execution, or with the `EXPLAIN` or `EXPLAIN ANALYZE` commands.

The parameter can be set for a database system, an individual database, or a session.

Value Range	Default	Set Classifications
Boolean	true	master session reload

optimizer_print_optimization_stats

When GPORCA is enabled (the default), this parameter enables logging of GPORCA query optimization statistics for various optimization stages for a query. The default value is `off`, do not log optimization statistics. To log the optimization statistics, this parameter must be set to `on` and the parameter `client_min_messages` must be set to `log`.

- `set optimizer_print_optimization_stats = on;`
- `set client_min_messages = 'log';`

The information is logged during query execution, or with the `EXPLAIN` or `EXPLAIN ANALYZE` commands.

This parameter can be set for a database system, an individual database, or a session or query.

Value Range	Default	Set Classifications
Boolean	off	master session reload

optimizer_sort_factor

When GPORCA is enabled (the default), `optimizer_sort_factor` controls the cost factor to apply to sorting operations during query optimization. The default value `1` specifies the default sort cost factor. The value is a ratio of increase or decrease from the default factor. For example, a value of `2.0` sets the cost factor at twice the default, and a value of `0.5` sets the factor at half the default.

The parameter can be set for a database system, an individual database, or a session or query.

Value Range	Default	Set Classifications
Decimal > 0	1	master session reload

optimizer_use_gpdb_allocators

When GPORCA is enabled (the default) and this parameter is `true` (the default), GPORCA uses Greenplum Database memory management when executing queries. When set to `false`, GPORCA uses GPORCA-specific memory management. Greenplum Database memory management allows for faster optimization, reduced memory usage during optimization, and improves GPORCA support of vmem limits when compared to GPORCA-specific memory management.

For information about GPORCA, see *About GPORCA* in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
Boolean	true	master system restart

password_encryption

When a password is specified in `CREATE USER` or `ALTER USER` without writing either `ENCRYPTED` or `UNENCRYPTED`, this option determines whether the password is to be encrypted.

Value Range	Default	Set Classifications
Boolean	on	master session reload

password_hash_algorithm

Specifies the cryptographic hash algorithm that is used when storing an encrypted Greenplum Database user password. The default algorithm is MD5.

For information about setting the password hash algorithm to protect user passwords, see "Protecting Passwords in Greenplum Database" in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
MD5 SHA-256	MD5	master session reload superuser

plan_cache_mode

Prepared statements (either explicitly prepared or implicitly generated, for example by PL/pgSQL) can be executed using *custom* or *generic* plans. Custom plans are created for each execution using its specific set of parameter values, while generic plans do not rely on the parameter values and can be re-used across executions. The use of a generic plan saves planning time, but if the ideal plan depends strongly on the parameter values, then a generic plan might be inefficient. The choice between these options is normally made automatically, but it can be overridden by setting the `plan_cache_mode` parameter. If the prepared statement has no parameters, a generic plan is always used.

The allowed values are `auto` (the default), `force_custom_plan` and `force_generic_plan`. This setting is considered when a cached plan is to be executed, not when it is prepared. For more information see *PREPARE*.

The parameter can be set for a database system, an individual database, a session, or a query.

Value Range	Default	Set Classifications
auto force_custom_plan force_generic_plan	auto	master session reload

pljava_classpath

A colon (`:`) separated list of jar files or directories containing jar files needed for PL/Java functions. The full path to the jar file or directory must be specified, except the path can be omitted for jar files in the `$GPHOME/lib/postgresql/java` directory. The jar files must be installed in the same locations on all Greenplum hosts and readable by the `gpadmin` user.

The `pljava_classpath` parameter is used to assemble the PL/Java classpath at the beginning of each user session. Jar files added after a session has started are not available to that session.

If the full path to a jar file is specified in `pljava_classpath` it is added to the PL/Java classpath. When a directory is specified, any jar files the directory contains are added to the PL/Java classpath. The search

does not descend into subdirectories of the specified directories. If the name of a jar file is included in `pljava_classpath` with no path, the jar file must be in the `$GPHOME/lib/postgresql/java` directory.

> **Note:** Performance can be affected if there are many directories to search or a large number of jar files.

If *`pljava_classpath_insecure`* is `false`, setting the `pljava_classpath` parameter requires superuser privilege. Setting the classpath in SQL code will fail when the code is executed by a user without superuser privilege. The `pljava_classpath` parameter must have been set previously by a superuser or in the `postgresql.conf` file. Changing the classpath in the `postgresql.conf` file requires a reload (`gpstop -u`).

Value Range	Default	Set Classifications
string		master session reload superuser

pljava_classpath_insecure

Controls whether the server configuration parameter *`pljava_classpath`* can be set by a user without Greenplum Database superuser privileges. When `true`, `pljava_classpath` can be set by a regular user. Otherwise, *`pljava_classpath`* can be set only by a database superuser. The default is `false`.

> **Warning:** Enabling this parameter exposes a security risk by giving non-administrator database users the ability to run unauthorized Java methods.

Value Range	Default	Set Classifications
Boolean	false	master session restart superuser

pljava_statement_cache_size

Sets the size in KB of the JRE MRU (Most Recently Used) cache for prepared statements.

Value Range	Default	Set Classifications
number of kilobytes	10	master system restart superuser

pljava_release_lingering_savepoints

If true, lingering savepoints used in PL/Java functions will be released on function exit. If false, savepoints will be rolled back.

Value Range	Default	Set Classifications
Boolean	true	master system restart superuser

pljava_vmoptions

Defines the startup options for the Java VM. The default value is an empty string ("").

Value Range	Default	Set Classifications
string		master system reload superuser

port

The database listener port for a Greenplum instance. The master and each segment has its own port. Port numbers for the Greenplum system must also be changed in the `gp_segment_configuration` catalog. You must shut down your Greenplum Database system before changing port numbers.

Value Range	Default	Set Classifications
any valid port number	5432	local system restart

random_page_cost

Sets the estimate of the cost of a nonsequentially fetched disk page for the Postgres Planner. This is measured as a multiple of the cost of a sequential page fetch. A higher value makes it more likely a sequential scan will be used, a lower value makes it more likely an index scan will be used.

Value Range	Default	Set Classifications
floating point	100	master session reload

readable_external_table_timeout

When an SQL query reads from an external table, the parameter value specifies the amount of time in seconds that Greenplum Database waits before cancelling the query when data stops being returned from the external table.

The default value of 0, specifies no time out. Greenplum Database does not cancel the query.

If queries that use gpfdist run a long time and then return the error "intermittent network connectivity issues", you can specify a value for readable_external_table_timeout. If no data is returned by gpfdist for the specified length of time, Greenplum Database cancels the query.

Value Range	Default	Set Classifications
integer >= 0	0	master system reload

repl_catchup_within_range

For Greenplum Database master mirroring, controls updates to the active master. If the number of WAL segment files that have not been processed by the `walsender` exceeds this value, Greenplum Database updates the active master.

If the number of segment files does not exceed the value, Greenplum Database blocks updates to the to allow the `walsender` process the files. If all WAL segments have been processed, the active master is updated.

Value Range	Default	Set Classifications
0 - 64	1	master system reload superuser

replication_timeout

For Greenplum Database master mirroring, sets the maximum time in milliseconds that the `walsender` process on the active master waits for a status message from the `walreceiver` process on the standby master. If a message is not received, the `walsender` logs an error message.

The *wal_receiver_status_interval* controls the interval between `walreceiver` status messages.

Value Range	Default	Set Classifications
0 - INT_MAX	60000 ms (60 seconds)	master system reload superuser

regex_flavor

The 'extended' setting may be useful for exact backwards compatibility with pre-7.4 releases of PostgreSQL.

Value Range	Default	Set Classifications
advanced extended basic	advanced	master session reload

resource_cleanup_gangs_on_wait

Note: The `resource_cleanup_gangs_on_wait` server configuration parameter is enforced only when resource queue-based resource management is active.

If a statement is submitted through a resource queue, clean up any idle query executor worker processes before taking a lock on the resource queue.

Value Range	Default	Set Classifications
Boolean	on	master system restart

resource_select_only

Note: The `resource_select_only` server configuration parameter is enforced only when resource queue-based resource management is active.

Sets the types of queries managed by resource queues. If set to on, then `SELECT`, `SELECT INTO`, `CREATE TABLE AS SELECT`, and `DECLARE CURSOR` commands are evaluated. If set to off `INSERT`, `UPDATE`, and `DELETE` commands will be evaluated as well.

Value Range	Default	Set Classifications
Boolean	off	master system restart

runaway_detector_activation_percent

For queries that are managed by resource queues or resource groups, this parameter determines when Greenplum Database terminates running queries based on the amount of memory the queries are using. A value of 100 disables the automatic termination of queries based on the percentage of memory that is utilized.

Either the resource queue or the resource group management scheme can be active in Greenplum Database; both schemes cannot be active at the same time. The server configuration parameter *`gp_resource_manager`* controls which scheme is active.

When resource queues are enabled - This parameter sets the percent of utilized Greenplum Database vmem memory that triggers the termination of queries. If the percentage of vmem memory that is utilized for a Greenplum Database segment exceeds the specified value, Greenplum Database terminates queries managed by resource queues based on memory usage, starting with the query consuming the largest amount of memory. Queries are terminated until the percentage of utilized vmem is below the specified percentage.

Specify the maximum vmem value for active Greenplum Database segment instances with the server configuration parameter *`gp_vmem_protect_limit`*.

For example, if vmem memory is set to 10GB, and this parameter is 90 (90%), Greenplum Database starts terminating queries when the utilized vmem memory exceeds 9 GB.

For information about resource queues, see *Using Resource Queues*.

When resource groups are enabled - This parameter sets the percent of utilized resource group global shared memory that triggers the termination of queries that are managed by resource groups that are

configured to use the `vmtracker` memory auditor, such as `admin_group` and `default_group`. For information about memory auditors, see *Memory Auditor*.

Resource groups have a global shared memory pool when the sum of the `MEMORY_LIMIT` attribute values configured for all resource groups is less than 100. For example, if you have 3 resource groups configured with `memory_limit` values of 10 , 20, and 30, then global shared memory is 40% = 100% - (10% + 20% + 30%). See *Global Shared Memory*.

If the percentage of utilized global shared memory exceeds the specified value, Greenplum Database terminates queries based on memory usage, selecting from queries managed by the resource groups that are configured to use the `vmtracker` memory auditor. Greenplum Database starts with the query consuming the largest amount of memory. Queries are terminated until the percentage of utilized global shared memory is below the specified percentage.

For example, if global shared memory is 10GB, and this parameter is 90 (90%), Greenplum Database starts terminating queries when the utilized global shared memory exceeds 9 GB.

For information about resource groups, see *Using Resource Groups*.

Value Range	Default	Set Classifications
percentage (integer)	90	local system restart

search_path

Specifies the order in which schemas are searched when an object is referenced by a simple name with no schema component. When there are objects of identical names in different schemas, the one found first in the search path is used. The system catalog schema, *pg_catalog*, is always searched, whether it is mentioned in the path or not. When objects are created without specifying a particular target schema, they will be placed in the first schema listed in the search path. The current effective value of the search path can be examined via the SQL function *current_schemas()*. *current_schemas()* shows how the requests appearing in *search_path* were resolved.

Value Range	Default	Set Classifications
a comma-separated list of schema names	$user,public	master session reload

seq_page_cost

For the Postgres Planner, sets the estimate of the cost of a disk page fetch that is part of a series of sequential fetches.

Value Range	Default	Set Classifications
floating point	1	master session reload

server_encoding

Reports the database encoding (character set). It is determined when the Greenplum Database array is initialized. Ordinarily, clients need only be concerned with the value of *client_encoding*.

Value Range	Default	Set Classifications
<system dependent>	UTF8	read only

server_version

Reports the version of PostgreSQL that this release of Greenplum Database is based on.

Value Range	Default	Set Classifications
string	9.4.20	read only

server_version_num

Reports the version of PostgreSQL that this release of Greenplum Database is based on as an integer.

Value Range	Default	Set Classifications
integer	90420	read only

shared_buffers

Sets the amount of memory a Greenplum Database segment instance uses for shared memory buffers. This setting must be at least 128KB and at least 16KB times *max_connections*.

Each Greenplum Database segment instance calculates and attempts to allocate certain amount of shared memory based on the segment configuration. The value of `shared_buffers` is significant portion of this shared memory calculation, but is not all it. When setting `shared_buffers`, the values for the operating system parameters `SHMMAX` or `SHMALL` might also need to be adjusted.

The operating system parameter `SHMMAX` specifies maximum size of a single shared memory allocation. The value of `SHMMAX` must be greater than this value:

```
shared_buffers + other_seg_shmem
```

The value of *other_seg_shmem* is the portion the Greenplum Database shared memory calculation that is not accounted for by the `shared_buffers` value. The *other_seg_shmem* value will vary based on the segment configuration.

With the default Greenplum Database parameter values, the value for *other_seg_shmem* is approximately 111MB for Greenplum Database segments and approximately 79MB for the Greenplum Database master.

The operating system parameter `SHMALL` specifies the maximum amount of shared memory on the host. The value of `SHMALL` must be greater than this value:

```
(num_instances_per_host * ( shared_buffers + other_seg_shmem ))
+ other_app_shared_mem
```

The value of *other_app_shared_mem* is the amount of shared memory that is used by other applications and processes on the host.

When shared memory allocation errors occur, possible ways to resolve shared memory allocation issues are to increase `SHMMAX` or `SHMALL`, or decrease `shared_buffers` or `max_connections`.

See the *Greenplum Database Installation Guide* for information about the Greenplum Database values for the parameters `SHMMAX` and `SHMALL`.

Value Range	Default	Set Classifications
integer > 16K * *max_connections*	125MB	local system restart

shared_preload_libraries

A comma-separated list of shared libraries that are to be preloaded at server start. PostgreSQL procedural language libraries can be preloaded in this way, typically by using the syntax '`$libdir/plXXX`' where *XXX* is pgsql, perl, tcl, or python. By preloading a shared library, the library startup time is avoided when the library is first used. If a specified library is not found, the server will fail to start.

> **Note:** When you add a library to `shared_preload_libraries`, be sure to retain any previous setting of the parameter.

Value Range	Default	Set Classifications
		local system restart

ssl

Enables SSL connections.

Value Range	Default	Set Classifications
Boolean	off	master system restart

ssl_ciphers

Specifies a list of SSL ciphers that are allowed to be used on secure connections. `ssl_ciphers` *overrides* any ciphers string specified in `/etc/openssl.cnf`. The default value `ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH` enables all ciphers except for ADH, LOW, EXP, and MD5 ciphers, and prioritizes ciphers by their strength.

> **Note:** With TLS 1.2 some ciphers in MEDIUM and HIGH strength still use NULL encryption (no encryption for transport), which the default `ssl_ciphers` string allows. To bypass NULL ciphers with TLS 1.2 use a string such as `TLSv1.2:!eNULL:!aNULL`.

See the openssl manual page for a list of supported ciphers.

Value Range	Default	Set Classifications
string	ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH	master system restart

standard_conforming_strings

Determines whether ordinary string literals ('...') treat backslashes literally, as specified in the SQL standard. The default value is on. Turn this parameter off to treat backslashes in string literals as escape characters instead of literal backslashes. Applications may check this parameter to determine how string literals are processed. The presence of this parameter can also be taken as an indication that the escape string syntax (E'...') is supported.

Value Range	Default	Set Classifications
Boolean	on	master session reload

statement_mem

Allocates segment host memory per query. The amount of memory allocated with this parameter cannot exceed *max_statement_mem* or the memory limit on the resource queue or resource group through which the query was submitted. If additional memory is required for a query, temporary spill files on disk are used.

If you are using resource groups to control resource allocation in your Greenplum Database cluster:

- Greenplum Database uses `statement_mem` to control query memory usage when the resource group `MEMORY_SPILL_RATIO` is set to 0.
- You can use the following calculation to estimate a reasonable `statement_mem` value:

```
rg_perseg_mem = ((RAM * (vm.overcommit_ratio / 100) + SWAP) *
 gp_resource_group_memory_limit) / num_active_primary_segments
statement_mem = rg_perseg_mem / max_expected_concurrent_queries
```

If you are using resource queues to control resource allocation in your Greenplum Database cluster:

- When *gp_resqueue_memory_policy* `=auto`, `statement_mem` and resource queue memory limits control query memory usage.
- You can use the following calculation to estimate a reasonable `statement_mem` value for a wide variety of situations:

```
( gp_vmem_protect_limitGB * .9 ) / max_expected_concurrent_queries
```

  For example, with a *gp_vmem_protect_limit* set to 8192MB (8GB) and assuming a maximum of 40 concurrent queries with a 10% buffer, you would use the following calculation to determine the `statement_mem` value:

```
(8GB * .9) / 40 = .18GB = 184MB
```

When changing both `max_statement_mem` and `statement_mem`, `max_statement_mem` must be changed first, or listed first in the `postgresql.conf` file.

Value Range	Default	Set Classifications
number of kilobytes	128MB	master session reload

statement_timeout

Abort any statement that takes over the specified number of milliseconds. 0 turns off the limitation.

Value Range	Default	Set Classifications
number of milliseconds	0	master session reload

stats_queue_level

> **Note:** The `stats_queue_level` server configuration parameter is enforced only when resource queue-based resource management is active.

Collects resource queue statistics on database activity.

Value Range	Default	Set Classifications
Boolean	off	master session reload

superuser_reserved_connections

Determines the number of connection slots that are reserved for Greenplum Database superusers.

Value Range	Default	Set Classifications
integer < *max_connections*	3	local system restart

tcp_keepalives_count

How many keepalives may be lost before the connection is considered dead. A value of 0 uses the system default. If TCP_KEEPCNT is not supported, this parameter must be 0.

Use this parameter for all connections that are not between a primary and mirror segment.

Value Range	Default	Set Classifications
number of lost keepalives	0	local system restart

tcp_keepalives_idle

Number of seconds between sending keepalives on an otherwise idle connection. A value of 0 uses the system default. If TCP_KEEPIDLE is not supported, this parameter must be 0.

Use this parameter for all connections that are not between a primary and mirror segment.

Value Range	Default	Set Classifications
number of seconds	0	local system restart

tcp_keepalives_interval

How many seconds to wait for a response to a keepalive before retransmitting. A value of 0 uses the system default. If TCP_KEEPINTVL is not supported, this parameter must be 0.

Use this parameter for all connections that are not between a primary and mirror segment.

Value Range	Default	Set Classifications
number of seconds	0	local system restart

temp_buffers

Sets the maximum memory, in blocks, to allow for temporary buffers by each database session. These are session-local buffers used only for access to temporary tables. The setting can be changed within individual sessions, but only up until the first use of temporary tables within a session. The cost of setting a large value in sessions that do not actually need a lot of temporary buffers is only a buffer descriptor for each block, or about 64 bytes, per increment. However if a buffer is actually used, an additional 32768 bytes will be consumed.

You can set this parameter to the number of 32K blocks (for example, `1024` to allow 32MB for buffers), or specify the maximum amount of memory to allow (for example `'48MB'` for 1536 blocks). The `gpconfig` utility and `SHOW` command report the maximum amount of memory allowed for temporary buffers.

Value Range	Default	Set Classifications
integer	1024 (32MB)	master session reload

temp_tablespaces

Specifies tablespaces in which to create temporary objects (temp tables and indexes on temp tables) when a `CREATE` command does not explicitly specify a tablespace. Temporary files for purposes such as sorting large data sets are also created in these tablespaces.

The value is a comma-separated list of names of tablespaces. When there is more than one name in the list, Greenplum chooses a random member of the list each time a temporary object is to be created; except that within a transaction, successively created temporary objects are placed in successive tablespaces from the list. If the selected element of the list is an empty string, Greenplum automatically uses the default tablespace of the current database instead.

When `temp_tablespaces` is set interactively, specifying a nonexistent tablespace is an error, as is specifying a tablespace for which the user does not have `CREATE` privilege. However, when using a previously set value, nonexistent tablespaces are ignored, as are tablespaces for which the user lacks `CREATE` privilege. In particular, this rule applies when using a value set in `postgresql.conf`.

The default value is an empty string, which results in all temporary objects being created in the default tablespace of the current database.

See also *default_tablespace*.

Value Range	Default	Set Classifications
one or more tablespace names	unset	master session reload

TimeZone

Sets the time zone for displaying and interpreting time stamps. The default is to use whatever the system environment specifies as the time zone. See *Date/Time Keywords* in the PostgreSQL documentation.

Value Range	Default	Set Classifications
time zone abbreviation		local restart

timezone_abbreviations

Sets the collection of time zone abbreviations that will be accepted by the server for date time input. The default is `Default`, which is a collection that works in most of the world. `Australia` and `India`, and other collections can be defined for a particular installation. Possible values are names of configuration files stored in `$GPHOME/share/postgresql/timezonesets/`.

To configure Greenplum Database to use a custom collection of timezones, copy the file that contains the timezone definitions to the directory `$GPHOME/share/postgresql/timezonesets/` on the Greenplum Database master and segment hosts. Then set value of the server configuration parameter `timezone_abbreviations` to the file. For example, to use a file `custom` that contains the default timezones and the WIB (Waktu Indonesia Barat) timezone.

1. Copy the file `Default` from the directory `$GPHOME/share/postgresql/timezonesets/` the file `custom`. Add the WIB timezone information from the file `Asia.txt` to the `custom`.
2. Copy the file `custom` to the directory `$GPHOME/share/postgresql/timezonesets/` on the Greenplum Database master and segment hosts.
3. Set value of the server configuration parameter `timezone_abbreviations` to `custom`.
4. Reload the server configuration file (`gpstop -u`).

Value Range	Default	Set Classifications
string	Default	master session reload

track_activities

Enables the collection of information on the currently executing command of each session, along with the time when that command began execution. The default value is `true`. Only superusers can change this setting. See the *`pg_stat_activity`* view.

> **Note:** Even when enabled, this information is not visible to all users, only to superusers and the user owning the session being reported on, so it should not represent a security risk.

Value Range	Default	Set Classifications
Boolean	true	master system reload superuser

track_activity_query_size

Sets the maximum length limit for the query text stored in `query` column of the system catalog view *pg_stat_activity*. The minimum length is 1024 characters.

Value Range	Default	Set Classifications
integer	1024	local system restart

track_counts

Enables the collection of information on the currently executing command of each session, along with the time at which that command began execution.

Value Range	Default	Set Classifications
Boolean	true	master session reload superuser

transaction_isolation

Sets the current transaction's isolation level.

Value Range	Default	Set Classifications
read committed serializable	read committed	master session reload

transaction_read_only

Sets the current transaction's read-only status.

Value Range	Default	Set Classifications
Boolean	off	master session reload

transform_null_equals

When on, expressions of the form expr = NULL (or NULL = expr) are treated as expr IS NULL, that is, they return true if expr evaluates to the null value, and false otherwise. The correct SQL-spec-compliant behavior of expr = NULL is to always return null (unknown).

Value Range	Default	Set Classifications
Boolean	off	master session reload

unix_socket_directories

Specifies the directory of the UNIX-domain socket on which the server is to listen for connections from client applications. Multiple sockets can be created by listing multiple directories separated by commas.

> **Important:** Do not change the value of this parameter. The default location is required for Greenplum Database utilities.

Value Range	Default	Set Classifications
directory path	unset	local system restart

unix_socket_group

Sets the owning group of the UNIX-domain socket. By default this is an empty string, which uses the default group for the current user.

Value Range	Default	Set Classifications
UNIX group name	unset	local system restart

unix_socket_permissions

Sets the access permissions of the UNIX-domain socket. UNIX-domain sockets use the usual UNIX file system permission set. Note that for a UNIX-domain socket, only write permission matters.

Value Range	Default	Set Classifications
numeric UNIX file permission mode (as accepted by the *chmod* or *umask* commands)	511	local system restart

update_process_title

Enables updating of the process title every time a new SQL command is received by the server. The process title is typically viewed by the `ps` command.

Value Range	Default	Set Classifications
Boolean	on	local system restart

vacuum_cost_delay

The length of time that the process will sleep when the cost limit has been exceeded. 0 disables the cost-based vacuum delay feature.

Value Range	Default	Set Classifications
milliseconds < 0 (in multiples of 10)	0	local system restart

vacuum_cost_limit

The accumulated cost that will cause the vacuuming process to sleep.

Value Range	Default	Set Classifications
integer > 0	200	local system restart

vacuum_cost_page_dirty

The estimated cost charged when vacuum modifies a block that was previously clean. It represents the extra I/O required to flush the dirty block out to disk again.

Value Range	Default	Set Classifications
integer > 0	20	local system restart

vacuum_cost_page_hit

The estimated cost for vacuuming a buffer found in the shared buffer cache. It represents the cost to lock the buffer pool, lookup the shared hash table and scan the content of the page.

Value Range	Default	Set Classifications
integer > 0	1	local system restart

vacuum_cost_page_miss

The estimated cost for vacuuming a buffer that has to be read from disk. This represents the effort to lock the buffer pool, lookup the shared hash table, read the desired block in from the disk and scan its content.

Value Range	Default	Set Classifications
integer > 0	10	local system restart

vacuum_freeze_min_age

Specifies the cutoff age (in transactions) that `VACUUM` should use to decide whether to replace transaction IDs with *FrozenXID* while scanning a table.

For information about `VACUUM` and transaction ID management, see "Managing Data" in the *Greenplum Database Administrator Guide* and the *PostgreSQL documentation*.

Value Range	Default	Set Classifications
integer 0-100000000000	100000000	local system restart

validate_previous_free_tid

Enables a test that validates the free tuple ID (TID) list. The list is maintained and used by Greenplum Database. Greenplum Database determines the validity of the free TID list by ensuring the previous free TID of the current free tuple is a valid free tuple. The default value is `true`, enable the test.

If Greenplum Database detects a corruption in the free TID list, the free TID list is rebuilt, a warning is logged, and a warning is returned by queries for which the check failed. Greenplum Database attempts to execute the queries.

Note: If a warning is returned, please contact Pivotal Support.

Value Range	Default	Set Classifications
Boolean	true	master session reload

verify_gpfdists_cert

When a Greenplum Database external table is defined with the `gpfdists` protocol to use SSL security, this parameter controls whether SSL certificate authentication is enabled. The default is `true`, SSL authentication is enabled when Greenplum Database communicates with the `gpfdist` utility to either read data from or write data to an external data source.

The value `false` disables SSL certificate authentication. These SSL exceptions are ignored:

- The self-signed SSL certificate that is used by `gpfdist` is not trusted by Greenplum Database.
- The host name contained in the SSL certificate does not match the host name that is running `gpfdist`.

You can set the value to `false` to disable authentication when testing the communication between the Greenplum Database external table and the `gpfdist` utility that is serving the external data.

> **Warning:** Disabling SSL certificate authentication exposes a security risk by not validating the `gpfdists` SSL certificate.

For information about the `gpfdists` protocol, see *gpfdists:// Protocol*. For information about running the `gpfdist` utility, see *gpfdist*.

Value Range	Default	Set Classifications
Boolean	true	master session reload

vmem_process_interrupt

Enables checking for interrupts before reserving vmem memory for a query during Greenplum Database query execution. Before reserving further vmem for a query, check if the current session for the query has a pending query cancellation or other pending interrupts. This ensures more responsive interrupt processing, including query cancellation requests. The default is `off`.

Value Range	Default	Set Classifications
Boolean	off	master session reload

wait_for_replication_threshold

When Greenplum Database segment mirroring is enabled, specifies the maximum amount of Write-Ahead Logging (WAL)-based records (in KB) written by a transaction on the primary segment instance before the records are written to the mirror segment instance for replication. As the default, Greenplum Database writes the records to the mirror segment instance when a checkpoint occurs or the `wait_for_replication_threshold` value is reached.

A value of 0 disables the check for the amount of records. The records are written to the mirror segment instance only after a checkpoint occurs.

If you set the value to 0, database performance issues might occur under heavy loads that perform long transactions that do not perform a checkpoint operation.

Value Range	Default	Set Classifications
0 - MAX-INT / 1024	1024	master system restart

wal_keep_segments

For Greenplum Database master mirroring, sets the maximum number of processed WAL segment files that are saved by the by the active Greenplum Database master if a checkpoint operation occurs.

The segment files are used to synchronize the active master on the standby master.

Value Range	Default	Set Classifications
integer	5	master system reload superuser

wal_receiver_status_interval

For Greenplum Database master mirroring, sets the interval in seconds between `walreceiver` process status messages that are sent to the active master. Under heavy loads, the time might be longer.

The value of *replication_timeout* controls the time that the `walsender` process waits for a `walreceiver` message.

Value Range	Default	Set Classifications
integer 0- INT_MAX/1000	10 sec	master system reload superuser

writable_external_table_bufsize

Size of the buffer (in KB) that Greenplum Database uses for network communication, such as the `gpfdist` utility and external web tables (that use http). Greenplum Database stores data in the buffer before writing the data out. For information about `gpfdist`, see the *Greenplum Database Utility Guide*.

Value Range	Default	Set Classifications
integer 32 - 131072 (32KB - 128MB)	64	local session reload

xid_stop_limit

The number of transaction IDs prior to the ID where transaction ID wraparound occurs. When this limit is reached, Greenplum Database stops creating new transactions to avoid data loss due to transaction ID wraparound.

Value Range	Default	Set Classifications
integer 10000000 - 2000000000	1000000000	local system restart

xid_warn_limit

The number of transaction IDs prior to the limit specified by *xid_stop_limit*. When Greenplum Database reaches this limit, it issues a warning to perform a VACUUM operation to avoid data loss due to transaction ID wraparound.

Value Range	Default	Set Classifications
integer 10000000 - 2000000000	500000000	local system restart

xmlbinary

Specifies how binary values are encoded in XML data. For example, when `bytea` values are converted to XML. The binary data can be converted to either base64 encoding or hexadecimal encoding. The default is base64.

The parameter can be set for a database system, an individual database, or a session.

Value Range	Default	Set Classifications
base64 hex	base64	master session reload

xmloption

Specifies whether XML data is to be considered as an XML document (`document`) or XML content fragment (`content`) for operations that perform implicit parsing and serialization. The default is `content`.

This parameter affects the validation performed by `xml_is_well_formed()`. If the value is `document`, the function checks for a well-formed XML document. If the value is `content`, the function checks for a well-formed XML content fragment.

> **Note:** An XML document that contains a document type declaration (DTD) is not considered a valid XML content fragment. If `xmloption` set to `content`, XML that contains a DTD is not considered valid XML.
>
> To cast a character string that contains a DTD to the `xml` data type, use the `xmlparse` function with the `document` keyword, or change the `xmloption` value to `document`.

The parameter can be set for a database system, an individual database, or a session. The SQL command to set this option for a session is also available in Greenplum Database.

```
SET XML OPTION { DOCUMENT | CONTENT }
```

Value Range	Default	Set Classifications
document content	content	master session reload

System Catalogs

This reference describes the Greenplum Database system catalog tables and views. System tables prefixed with gp_ relate to the parallel features of Greenplum Database. Tables prefixed with pg_ are either standard PostgreSQL system catalog tables supported in Greenplum Database, or are related to features Greenplum that provides to enhance PostgreSQL for data warehousing workloads. Note that the global system catalog for Greenplum Database resides on the master instance.

> **Warning:** Changes to Pivotal Greenplum Database system catalog tables or views are not supported. If a catalog table or view is changed by the customer, the Pivotal Greenplum Database cluster is not supported. The cluster must be reinitialized and restored by the customer.

System Tables

- *pg_foreign_table*
- *pg_index*
- *pg_inherits*
- *pg_language*
- *pg_largeobject*
- *pg_listener*
- *pg_namespace*
- *pg_opclass*
- *pg_operator*
- *pg_opfamily*
- *pg_partition*
- *pg_partition_rule*
- *pg_pltemplate*
- *pg_proc*
- *pg_resgroup*
- *pg_resgroupcapability*
- *pg_resourcetype*
- *pg_resqueue*
- *pg_resqueuecapability*
- *pg_rewrite*
- *pg_shdepend*
- *pg_shdescription*
- *pg_stat_last_operation*
- *pg_stat_last_shoperation*
- *pg_stat_replication*
- *pg_statistic*
- *pg_tablespace*
- *pg_trigger*
- *pg_type*
- *pg_user_mapping*

System Views

Greenplum Database provides the following system views not available in PostgreSQL.

- *gp_distributed_log*
- *gp_distributed_xacts*
- *gp_pgdatabase*
- *gp_resgroup_config*
- *gp_resgroup_status*
- *gp_resgroup_status_per_host*
- *gp_resgroup_status_per_segment*
- *gp_resqueue_status*
- *gp_transaction_log*
- *gpexpand.expansion_progress*
- *pg_matviews*
- *pg_max_external_files*
- *pg_partition_columns*
- *pg_partition_templates*
- *pg_partitions*
- *pg_resqueue_attributes*

- pg_resqueue_status (Deprecated. Use *gp_toolkit.gp_resqueue_status*.)
- *pg_stat_activity*
- *pg_stat_all_indexes*
- *pg_stat_all_tables*
- *pg_stat_replication*
- *pg_stat_resqueues*
- session_level_memory_consumption (See *Viewing Session Memory Usage Information*)

For more information about the standard system views supported in PostgreSQL and Greenplum Database, see the following sections of the PostgreSQL documentation:

- *System Views*
- *Statistics Collector Views*
- *The Information Schema*

System Catalogs Definitions

System catalog table and view definitions in alphabetical order.

foreign_data_wrapper_options

The `foreign_data_wrapper_options` view contains all of the otpions defined for foreign-data wrappers in the current database. Greenplum Database displays only those foreign-data wrappers to which the current user has access (by way of being the owner or having some privilege).

Table 116: foreign_data_wrapper_options

column	type	references	description
`foreign_data_wrapper_catalog`	sql_identifier		Name of the database in which the foreign-data wrapper is defined (always the current database).
`foreign_data_wrapper_name`	sql_identifier		Name of the foreign-data wrapper.
`option_name`	sql_identifier		Name of an option.
`option_value`	character_data		Value of the option.

foreign_data_wrappers

The `foreign_data_wrappers` view contains all foreign-data wrappers defined in the current database. Greenplum Database displays only those foreign-data wrappers to which the current user has access (by way of being the owner or having some privilege).

Table 117: foreign_data_wrappers

column	type	references	description
`foreign_data_wrapper_catalog`	sql_identifier		Name of the database in which the foreign-data wrapper is defined (always the current database).

column	type	references	description
foreign_data_wrapper_name	sql_identifier		Name of the foreign-data wrapper.
authorization_identifier	sql_identifier		Name of the owner of the foreign server.
library_name	character_data		File name of the library that implements this foreign-data wrapper.
foreign_data_wrapper_language	character_data		Language used to implement the foreign-data wrapper.

foreign_server_options

The foreign_server_options view contains all of the options defined for foreign servers in the current database. Greenplum Database displays only those foreign servers to which the current user has access (by way of being the owner or having some privilege).

Table 118: foreign_server_options

column	type	references	description
foreign_server_catalog	sql_identifier		Name of the database in which the foreign server is defined (always the current database).
foreign_server_name	sql_identifier		Name of the foreign server.
option_name	sql_identifier		Name of an option.
option_value	character_data		Value of the option.

foreign_servers

The foreign_servers view contains all foreign servers defined in the current database. Greenplum Database displays only those foreign servers to which the current user has access (by way of being the owner or having some privilege).

Table 119: foreign_servers

column	type	references	description
foreign_server_catalog	sql_identifier		Name of the database in which the foreign server is defined (always the current database).
foreign_server_name	sql_identifier		Name of the foreign server.

column	type	references	description
foreign_data_wrapper_catalog	sql_identifier		Name of the database in which the foreign-data wrapper used by the foreign server is defined (always the current database).
foreign_data_wrapper_name	sql_identifier		Name of the foreign-data wrapper used by the foreign server.
foreign_server_type	character_data		Foreign server type information, if specified upon creation.
foreign_server_version	character_data		Foreign server version information, if specified upon creation.
authorization_identifier	sql_identifier		Name of the owner of the foreign server.

foreign_table_options

The foreign_table_options view contains all of the options defined for foreign tables in the current database. Greenplum Database displays only those foreign tables to which the current user has access (by way of being the owner or having some privilege).

Table 120: foreign_table_options

column	type	references	description
foreign_table_catalog	sql_identifier		Name of the database in which the foreign table is defined (always the current database).
foreign_table_schema	sql_identifier		Name of the schema that contains the foreign table.
foreign_table_name	sql_identifier		Name of the foreign table.
option_name	sql_identifier		Name of an option.
option_value	character_data		Value of the option.

foreign_tables

The foreign_tables view contains all foreign tables defined in the current database. Greenplum Database displays only those foreign tables to which the current user has access (by way of being the owner or having some privilege).

Table 121: foreign_tables

column	type	references	description
`foreign_table_catalog`	sql_identifier		Name of the database in which the foreign table is defined (always the current database).
`foreign_table_schema`	sql_identifier		Name of the schema that contains the foreign table.
`foreign_table_name`	sql_identifier		Name of the foreign table.
`foreign_server_catalog`	sql_identifier		Name of the database in which the foreign server is defined (always the current database).
`foreign_server_name`	sql_identifier		Name of the foreign server.

gp_configuration_history

The `gp_configuration_history` table contains information about system changes related to fault detection and recovery operations. The `fts_probe` process logs data to this table, as do certain related management utilities such as `gprecoverseg` and `gpinitsystem`. For example, when you add a new segment and mirror segment to the system, records for these events are logged to `gp_configuration_history`.

The event descriptions stored in this table may be helpful for troubleshooting serious system issues in collaboration with Pivotal Support technicians.

This table is populated only on the master. This table is defined in the `pg_global` tablespace, meaning it is globally shared across all databases in the system.

Table 122: pg_catalog.gp_configuration_history

column	type	references	description
`time`	timestamp with time zone		Timestamp for the event recorded.
`dbid`	smallint	gp_segment _ configuration.dbid	System-assigned ID. The unique identifier of a segment (or master) instance.
`desc`	text		Text description of the event.

For information about `gprecoverseg` and `gpinitsystem`, see the Greenplum Database Utility Guide.

gp_distributed_log

The `gp_distributed_log` view contains status information about distributed transactions and their associated local transactions. A distributed transaction is a transaction that involves modifying data on the

segment instances. Greenplum's distributed transaction manager ensures that the segments stay in synch. This view allows you to see the status of distributed transactions.

Table 123: pg_catalog.gp_distributed_log

column	type	references	description
segment_id	smallint	gp_segment_ configuration.content	The content id if the segment. The master is always -1 (no content).
dbid	small_int	gp_segment_ configuration.dbid	The unique id of the segment instance.
distributed_xid	xid		The global transaction id.
distributed_id	text		A system assigned ID for a distributed transaction.
status	text		The status of the distributed transaction (Committed or Aborted).
local_transaction	xid		The local transaction ID.

gp_distributed_xacts

The gp_distributed_xacts view contains information about Greenplum Database distributed transactions. A distributed transaction is a transaction that involves modifying data on the segment instances. Greenplum's distributed transaction manager ensures that the segments stay in synch. This view allows you to see the currently active sessions and their associated distributed transactions.

Table 124: pg_catalog.gp_distributed_xacts

column	type	references	description
distributed_xid	xid		The transaction ID used by the distributed transaction across the Greenplum Database array.
distributed_id	text		The distributed transaction identifier. It has 2 parts — a unique timestamp and the distributed transaction number.
state	text		The current state of this session with regards to distributed transactions.
gp_session_id	int		The ID number of the Greenplum Database session associated with this transaction.

column	type	references	description
xmin_distributed _ snapshot	xid		The minimum distributed transaction number found among all open transactions when this transaction was started. It is used for MVCC distributed snapshot purposes.

gp_distribution_policy

The gp_distribution_policy table contains information about Greenplum Database tables and their policy for distributing table data across the segments. This table is populated only on the master. This table is not globally shared, meaning each database has its own copy of this table.

Table 125: pg_catalog.gp_distribution_policy

column	type	references	description
localoid	oid	pg_class.oid	The table object identifier (OID).
policytype	char		The table distribution policy: • p - Partitioned policy. Table data is distributed among segment instances. • r - Replicated policy. Table data is replicated on each segment instance.
numsegments	integer		The number of segment instances on which the table data is distributed.
distkey	int2vector	pg_attribute.attnum	The column number(s) of the distribution column(s).
distclass	oidvector	pg_opclass.oid	The operator class identifier(s) of the distribution column(s).

gpexpand.expansion_progress

The gpexpand.expansion_progress view contains information about the status of a system expansion operation. The view provides calculations of the estimated rate of table redistribution and estimated time to completion.

Status for specific tables involved in the expansion is stored in *gpexpand.status_detail*.

Table 126: gpexpand.expansion_progress

column	type	references	description
`name`	text		Name for the data field provided. Includes: Bytes Left Bytes Done Estimated Expansion Rate Estimated Time to Completion Tables Expanded Tables Left
`value`	text		The value for the progress data. For example: `Estimated Expansion Rate - 9.75667095996092 MB/s`

gpexpand.status

The `gpexpand.status` table contains information about the status of a system expansion operation. Status for specific tables involved in the expansion is stored in *gpexpand.status_detail*.

In a normal expansion operation it is not necessary to modify the data stored in this table.

Table 127: gpexpand.status

column	type	references	description

`status`	text		Tracks the status of an expansion operation. Valid values are:
			`SETUP`
			`SETUP DONE`
			`EXPANSION STARTED`
			`EXPANSION STOPPED`
			`COMPLETED`

`updated`	timestamp without time zone		Timestamp of the last change in status.

gpexpand.status_detail

The `gpexpand.status_detail` table contains information about the status of tables involved in a system expansion operation. You can query this table to determine the status of tables being expanded, or to view the start and end time for completed tables.

This table also stores related information about the table such as the oid and disk size. Overall status information for the expansion is stored in *gpexpand.status*.

In a normal expansion operation it is not necessary to modify the data stored in this table.

Table 128: gpexpand.status_detail

column	type	references	description
`dbname`	text		Name of the database to which the table belongs.
`fq_name`	text		Fully qualified name of the table.
`table_oid`	oid		OID of the table.
`root_partition_name`	text		For a partitioned table, the name of the root partition. Otherwise, `None`.
`rank`	int		Rank determines the order in which tables are expanded. The expansion utility will sort on rank and expand the lowest-ranking tables first.
`external_writable`	boolean		Identifies whether or not the table is an external writable table. (External writable tables require a different syntax to expand).
`status`	text		Status of expansion for this table. Valid values are: NOT STARTED IN PROGRESS COMPLETED NO LONGER EXISTS
`expansion_started`	timestamp without time zone		Timestamp for the start of the expansion of this table. This field is only populated after a table is successfully expanded.
`expansion_finished`	timestamp without time zone		Timestamp for the completion of expansion of this table.

column	type	references	description
source_bytes			The size of disk space associated with the source table. Due to table bloat in heap tables and differing numbers of segments after expansion, it is not expected that the final number of bytes will equal the source number. This information is tracked to help provide progress measurement to aid in duration estimation for the end-to-end expansion operation.

gp_fastsequence

The gp_fastsequence table contains information about append-optimized and column-oriented tables. The last_sequence value indicates maximum row number currently used by the table.

Table 129: pg_catalog.gp_fastsequence

column	type	references	description
objid	oid	pg_class.oid	Object id of the pg_aoseg.pg_aocsseg_* table used to track append-optimized file segments.
objmod	bigint		Object modifier.
last_sequence	bigint		The last sequence number used by the object.

gp_id

The gp_id system catalog table identifies the Greenplum Database system name and number of segments for the system. It also has local values for the particular database instance (segment or master) on which the table resides. This table is defined in the pg_global tablespace, meaning it is globally shared across all databases in the system.

Table 130: pg_catalog.gp_id

column	type	references	description
gpname	name		The name of this Greenplum Database system.

column	type	references	description
numsegments	integer		The number of segments in the Greenplum Database system.
dbid	integer		The unique identifier of this segment (or master) instance.
content	integer		The ID for the portion of data on this segment instance. A primary and its mirror will have the same content ID. For a segment the value is from 0-*N-1*, where *N* is the number of segments in Greenplum Database. For the master, the value is -1.

gp_pgdatabase

The gp_pgdatabase view shows status information about the Greenplum segment instances and whether they are acting as the mirror or the primary. This view is used internally by the Greenplum fault detection and recovery utilities to determine failed segments.

Table 131: pg_catalog.gp_pgdatabase

column	type	references	description
dbid	smallint	gp_segment_configuration.dbid	System-assigned ID. The unique identifier of a segment (or master) instance.
isprimary	boolean	gp_segment_configuration.role	Whether or not this instance is active. Is it currently acting as the primary segment (as opposed to the mirror).
content	smallint	gp_segment_configuration.content	The ID for the portion of data on an instance. A primary segment instance and its mirror will have the same content ID. For a segment the value is from 0-*N-1*, where *N* is the number of segments in Greenplum Database. For the master, the value is -1.

column	type	references	description
definedprimary	boolean	gp_segment_configuration.preferred_role	Whether or not this instance was defined as the primary (as opposed to the mirror) at the time the system was initialized.

gp_resgroup_config

The gp_toolkit.gp_resgroup_config view allows administrators to see the current CPU, memory, and concurrency limits for a resource group.

> **Note:** The gp_resgroup_config view is valid only when resource group-based resource management is active.

Table 132: gp_toolkit.gp_resgroup_config

column	type	references	description
groupid	oid	pg_resgroup.oid	The ID of the resource group.
groupname	name	pg_resgroup.rsgname	The name of the resource group.
concurrency	text	pg_resgroupcapability.value for pg_resgroupcapability.reslimittype = 1	The concurrency (CONCURRENCY) value specified for the resource group.
cpu_rate_limit	text	pg_resgroupcapability.value for pg_resgroupcapability.reslimittype = 2	The CPU limit (CPU_RATE_LIMIT) value specified for the resource group, or -1.
memory_limit	text	pg_resgroupcapability.value for pg_resgroupcapability.reslimittype = 3	The memory limit (MEMORY_LIMIT) value specified for the resource group.
memory_shared_quota	text	pg_resgroupcapability.value for pg_resgroupcapability.reslimittype = 4	The shared memory quota (MEMORY_SHARED_QUOTA) value specified for the resource group.
memory_spill_ratio	text	pg_resgroupcapability.value for pg_resgroupcapability.reslimittype = 5	The memory spill ratio (MEMORY_SPILL_RATIO) value specified for the resource group.
memory_auditor	text	pg_resgroupcapability.value for pg_resgroupcapability.reslimittype = 6	The memory auditor in use for the resource group.

column	type	references	description
cpuset	text	pg_resgroupcapability. value for pg_ resgroupcapability. reslimittype = 7	The CPU cores reserved for the resource group, or -1.

gp_resgroup_status

The gp_toolkit.gp_resgroup_status view allows administrators to see status and activity for a resource group. It shows how many queries are waiting to run and how many queries are currently active in the system for each resource group. The view also displays current memory and CPU usage for the resource group.

> **Note:** The gp_resgroup_status view is valid only when resource group-based resource management is active.

Table 133: gp_toolkit.gp_resgroup_status

column	type	references	description
rsgname	name	pg_resgroup.rsgname	The name of the resource group.
groupid	oid	pg_resgroup.oid	The ID of the resource group.
num_running	integer		The number of transactions currently executing in the resource group.
num_queueing	integer		The number of currently queued transactions for the resource group.
num_queued	integer		The total number of queued transactions for the resource group since the Greenplum Database cluster was last started, excluding the num_queueing.
num_executed	integer		The total number of executed transactions in the resource group since the Greenplum Database cluster was last started, excluding the num_running.
total_queue_ duration	interval		The total time any transaction was queued since the Greenplum Database cluster was last started.

column	type	references	description
cpu_usage	json		A set of key-value pairs. For each segment instance (the key), the value is the real-time, per-segment instance CPU core usage by a resource group. The value is the sum of the percentages (as a decimal value) of CPU cores that are used by the resource group for the segment instance.
memory_usage	json		The real-time memory usage of the resource group on each Greenplum Database segment's host.

The `cpu_usage` field is a JSON-formatted, key:value string that identifies, for each resource group, the per-segment instance CPU core usage. The key is the segment id. The value is the sum of the percentages (as a decimal value) of the CPU cores used by the segment instance's resource group on the segment host; the maximum value is 1.00. The total CPU usage of all segment instances running on a host should not exceed the `gp_resource_group_cpu_limit`. Example `cpu_usage` column output:

```
{"-1":0.01, "0":0.31, "1":0.31}
```

In the example, segment `0` and segment `1` are running on the same host; their CPU usage is the same.

The `memory_usage` field is also a JSON-formatted, key:value string. The string contents differ depending upon the type of resource group. For each resource group that you assign to a role (default memory auditor `vmtracker`), this string identifies the used and available fixed and shared memory quota allocations on each segment. The key is segment id. The values are memory values displayed in MB units. The following example shows `memory_usage` column output for a single segment for a resource group that you assign to a role:

```
"0":{"used":0, "available":76, "quota_used":-1, "quota_available":60,
 "shared_used":0, "shared_available":16}
```

For each resource group that you assign to an external component, the `memory_usage` JSON-formatted string identifies the memory used and the memory limit on each segment. The following example shows `memory_usage` column output for an external component resource group for a single segment:

```
"1":{"used":11, "limit_granted":15}
```

gp_resgroup_status_per_host

The `gp_toolkit.gp_resgroup_status_per_host` view allows administrators to see current memory and CPU usage and allocation for each resource group on a per-host basis.

Memory amounts are specified in MBs.

Note: The `gp_resgroup_status_per_host` view is valid only when resource group-based resource management is active.

Table 134: gp_toolkit.gp_resgroup_status_per_host

column	type	references	description
`rsgname`	name	pg_resgroup.rsgname	The name of the resource group.
`groupid`	oid	pg_resgroup.oid	The ID of the resource group.
`hostname`	text	gp_segment_ configuration.hostname	The hostname of the segment host.
`cpu`	numeric		The real-time CPU core usage by the resource group on a host. The value is the sum of the percentages (as a decimal value) of the CPU cores that are used by the resource group on the host.
`memory_used`	integer		The real-time memory usage of the resource group on the host. This total includes resource group fixed and shared memory. It also includes global shared memory used by the resource group.
`memory_available`	integer		The unused fixed and shared memory for the resource group that is available on the host. This total does not include available resource group global shared memory.
`memory_quota_used`	integer		The real-time fixed memory usage for the resource group on the host.
`memory_quota_ available`	integer		The fixed memory available to the resource group on the host.

column	type	references	description
memory_shared_used	integer		The group shared memory used by the resource group on the host. If any global shared memory is used by the resource group, this amount is included in the total as well.
memory_shared_available	integer		The amount of group shared memory available to the resource group on the host. Resource group global shared memory is not included in this total.

gp_resgroup_status_per_segment

The gp_toolkit.gp_resgroup_status_per_segment view allows administrators to see current memory and CPU usage and allocation for each resource group on a per-host and per-segment basis.

Memory amounts are specified in MBs.

> **Note:** The gp_resgroup_status_per_segment view is valid only when resource group-based resource management is active.

Table 135: gp_toolkit.gp_resgroup_status_per_segment

column	type	references	description
rsgname	name	pg_resgroup.rsgname	The name of the resource group.
groupid	oid	pg_resgroup.oid	The ID of the resource group.
hostname	text	gp_segment_configuration.hostname	The hostname of the segment host.
segment_id	smallint	gp_segment_configuration.content	The content ID for a segment instance on the segment host.
cpu	numeric		The real-time, per-segment instance CPU core usage by the resource group on the host. The value is the sum of the percentages (as a decimal value) of the CPU cores that are used by the resource group for the segment instance.

column	type	references	description
memory_used	integer		The real-time memory usage of the resource group for the segment instance on the host. This total includes resource group fixed and shared memory. It also includes global shared memory used by the resource group.
memory_available	integer		The unused fixed and shared memory for the resource group for the segment instance on the host.
memory_quota_used	integer		The real-time fixed memory usage for the resource group for the segment instance on the host.
memory_quota_available	integer		The fixed memory available to the resource group for the segment instance on the host.
memory_shared_used	integer		The group shared memory used by the resource group for the segment instance on the host.
memory_shared_available	integer		The amount of group shared memory available for the segment instance on the host. Resource group global shared memory is not included in this total.

gp_resqueue_status

The gp_toolkit.gp_resqueue_status view allows administrators to see status and activity for a resource queue. It shows how many queries are waiting to run and how many queries are currently active in the system from a particular resource queue.

> **Note:** The gp_resqueue_status view is valid only when resource queue-based resource management is active.

Table 136: gp_toolkit.gp_resqueue_status

column	type	references	description
queueid	oid	gp_toolkit.gp_resqueue_queueid	The ID of the resource queue.

column	type	references	description
rsqname	name	gp_toolkit.gp_resqueue_rsqname	The name of the resource queue.
rsqcountlimit	real	gp_toolkit.gp_resqueue_rsqcountlimit	The active query threshold of the resource queue. A value of -1 means no limit.
rsqcountvalue	real	gp_toolkit.gp_resqueue_rsqcountvalue	The number of active query slots currently being used in the resource queue.
rsqcostlimit	real	gp_toolkit.gp_resqueue_rsqcostlimit	The query cost threshold of the resource queue. A value of -1 means no limit.
rsqcostvalue	real	gp_toolkit.gp_resqueue_rsqcostvalue	The total cost of all statements currently in the resource queue.
rsqmemorylimit	real	gp_toolkit.gp_resqueue_rsqmemorylimit	The memory limit for the resource queue.
rsqmemoryvalue	real	gp_toolkit.gp_resqueue_rsqmemoryvalue	The total memory used by all statements currently in the resource queue.
rsqwaiters	integer	gp_toolkit.gp_resqueue_rsqwaiter	The number of statements currently waiting in the resource queue.
rsqholders	integer	gp_toolkit.gp_resqueue_rsqholders	The number of statements currently running on the system from this resource queue.

gp_stat_replication

The gp_stat_replication view contains replication statistics of the walsender process that is used for Greenplum Database Write-Ahead Logging (WAL) replication when master or segment mirroring is enabled.

Table 137: gp_catalog.gp_stat_replication

column	type	references	description
gp_segment_id	integer		Unique identifier of a segment (or master) instance.
pid	integer		Process ID of the walsender backend process.

column	type	references	description
usesysid	oid		User system ID that runs the walsender backend process.
usename	name		User name that runs the walsender backend process.
application_name	text		Client application name.
client_addr	inet		Client IP address.
client_hostname	text		Client host name.
client_port	integer		Client port number.
backend_start	timestamp		Operation start timestamp.
state	text		walsender state. The value can be: startup backup catchup streaming
sent_location	text		walsender xlog record sent location.
write_location	text		walreceiver xlog record write location.
flush_location	text		walreceiver xlog record flush location.
replay_location	text		Master standby or segment mirror xlog record replay location.
sync_priority	integer		Priority. The value is 1.
sync_state	text		walsendersynchronization state. The value is sync.
sync_error	text		walsender synchronization error. none if no error.

gp_segment_configuration

The gp_segment_configuration table contains information about mirroring and segment instance configuration.

Table 138: pg_catalog.gp_segment_configuration

column	type	references	description
`dbid`	smallint		Unique identifier of a segment (or master) instance.
`content`	smallint		The content identifier for a segment instance. A primary segment instance and its corresponding mirror will always have the same content identifier. For a segment the value is from 0 to *N*-1, where *N* is the number of primary segments in the system. For the master, the value is always -1.
`role`	char		The role that a segment is currently running as. Values are p (primary) or `m`(mirror).
`preferred_role`	char		The role that a segment was originally assigned at initialization time. Values are p (primary) or `m` (mirror).

column	type	references	description
mode	char		The synchronization status of a segment instance with its mirror copy. Values are s (synchronized) or n (not synchronized). **Note:** This column always shows n for the master segment and s for the standby master segment, but these values do not describe the synchronization state for the master segment. Use *gp_stat_replication* to determine the synchronization state between the master and standby master.
status	char		The fault status of a segment instance. Values are u (up) or d (down).
port	integer		The TCP port the database server listener process is using.
hostname	text		The hostname of a segment host.
address	text		The hostname used to access a particular segment instance on a segment host. This value may be the same as hostname on systems that do not have per-interface hostnames configured.
datadir	text		Segment instance data directory.

gp_transaction_log

The gp_transaction_log view contains status information about transactions local to a particular segment. This view allows you to see the status of local transactions.

Table 139: pg_catalog.gp_transaction_log

column	type	references	description
segment_id	smallint	gp_segment_configuration.content	The content id if the segment. The master is always -1 (no content).
dbid	smallint	gp_segment_configuration.dbid	The unique id of the segment instance.
transaction	xid		The local transaction ID.
status	text		The status of the local transaction (Committed or Aborted).

gp_version_at_initdb

The gp_version_at_initdb table is populated on the master and each segment in the Greenplum Database system. It identifies the version of Greenplum Database used when the system was first initialized. This table is defined in the pg_global tablespace, meaning it is globally shared across all databases in the system.

Table 140: pg_catalog.gp_version

column	type	references	description
schemaversion	integer		Schema version number.
productversion	text		Product version number.

pg_aggregate

The pg_aggregate table stores information about aggregate functions. An aggregate function is a function that operates on a set of values (typically one column from each row that matches a query condition) and returns a single value computed from all these values. Typical aggregate functions are sum, count, and max. Each entry in pg_aggregate is an extension of an entry in pg_proc. The pg_proc entry carries the aggregate's name, input and output data types, and other information that is similar to ordinary functions.

Table 141: pg_catalog.pg_aggregate

column	type	references	description
aggfnoid	regproc	pg_proc.oid	OID of the aggregate function
aggkind	char		Aggregate kind: n for *normal* aggregates, o for *ordered-set* aggregates, or h for *hypothetical-set* aggregates

column	type	references	description
aggnumdirectargs	int2		Number of direct (non-aggregated) arguments of an ordered-set or hypothetical-set aggregate, counting a variadic array as one argument. If equal to pronargs, the aggregate must be variadic and the variadic array describes the aggregated arguments as well as the final direct arguments. Always zero for normal aggregates.
aggtransfn	regproc	pg_proc.oid	Transition function OID
aggfinalfn	regproc	pg_proc.oid	Final function OID (zero if none)
aggcombinefn	regproc	pg_proc.oid	Combine function OID (zero if none)
aggserialfn	regproc	pg_proc.oid	OID of the serialization function to convert transtype to bytea (zero if none)
aggdeserialfn	regproc	pg_proc.oid	OID of the deserialization function to convert bytea to transtype (zero if none)
aggmtransfn	regproc	pg_proc.oid	Forward transition function OID for moving-aggregate mode (zero if none)
aggminvtransfn	regproc	pg_proc.oid	Inverse transition function OID for moving-aggregate mode (zero if none)
aggmfinalfn	regproc	pg_proc.oid	Final function OID for moving-aggregate mode (zero if none)
aggfinalextra	bool		True to pass extra dummy arguments to aggfinalfn
aggmfinalextra	bool		True to pass extra dummy arguments to aggmfinalfn
aggsortop	oid	pg_operator.oid	Associated sort operator OID (zero if none)

column	type	references	description
aggtranstype	oid	pg_type.oid	Data type of the aggregate function's internal transition (state) data
aggtransspace	int4		Approximate average size (in bytes) of the transition state data, or zero to use a default estimate
aggmtranstype	oid	pg_type.oid	Data type of the aggregate function's internal transition (state) data for moving-aggregate mode (zero if none)
aggmtransspace	int4		Approximate average size (in bytes) of the transition state data for moving-aggregate mode, or zero to use a default estimate
agginitval	text		The initial value of the transition state. This is a text field containing the initial value in its external string representation. If this field is NULL, the transition state value starts out NULL.
aggminitval	text		The initial value of the transition state for moving- aggregate mode. This is a text field containing the initial value in its external string representation. If this field is NULL, the transition state value starts out NULL.

pg_am

The pg_am table stores information about index access methods. There is one row for each index access method supported by the system.

Table 142: pg_catalog.pg_am

column	type	references	description
`oid`	oid		Row identifier (hidden attribute; must be explicitly selected)
`amname`	name		Name of the access method
`amstrategies`	int2		Number of operator strategies for this access method, or zero if the access method does not have a fixed set of operator strategies
`amsupport`	int2		Number of support routines for this access method
`amcanorder`	boolean		Does the access method support ordered scans sorted by the indexed column's value?
`amcanorderbyop`	boolean		Does the access method support ordered scans sorted by the result of an operator on the indexed column?
`amcanbackward`	boolean		Does the access method support backward scanning?
`amcanunique`	boolean		Does the access method support unique indexes?
`amcanmulticol`	boolean		Does the access method support multicolumn indexes?
`amoptionalkey`	boolean		Does the access method support a scan without any constraint for the first index column?
`amsearcharray`	boolean		Does the access method support `ScalarArrayOpExpr` searches?
`amsearchnulls`	boolean		Does the access method support `IS NULL/NOT NULL` searches?
`amstorage`	boolean		Can index storage data type differ from column data type?

column	type	references	description
amclusterable	boolean		Can an index of this type be clustered on?
ampredlocks	boolean		Does an index of this type manage fine-grained predicae locks?
amkeytype	oid	pg_type.oid	Type of data stored in index, or zero if not a fixed type
aminsert	regproc	pg_proc.oid	"Insert this tuple" function
ambeginscan	regproc	pg_proc.oid	"Prepare for index scan" function
amgettuple	regproc	pg_proc.oid	"Next valid tuple" function, or zero if none
amgetbitmap	regproc	pg_proc.oid	"Fetch all tuples" function, or zero if none
amrescan	regproc	pg_proc.oid	"(Re)start index scan" function
amendscan	regproc	pg_proc.oid	"Clean up after index scan" function
ammarkpos	regproc	pg_proc.oid	"Mark current scan position" function
amrestrpos	regproc	pg_proc.oid	"Restore marked scan position" function
ambuild	regproc	pg_proc.oid	"Build new index" function
ambuildempty	regproc	pg_proc.oid	"Build empty index" function
ambulkdelete	regproc	pg_proc.oid	Bulk-delete function
amvacuumcleanup	regproc	pg_proc.oid	Post-VACUUM cleanup function
amcanreturn	regproc	pg_proc.oid	Function to check whether index supports index-only scans, or zero if none
amcostestimate	regproc	pg_proc.oid	Function to estimate cost of an index scan
amoptions	regproc	pg_proc.oid	Function to parse and validate reloptions for an index

pg_amop

The pg_amop table stores information about operators associated with index access method operator classes. There is one row for each operator that is a member of an operator class.

An entry's amopmethod must match the opfmethod of its containing operator family (including amopmethod here is an intentional denormalization of the catalog structure for performance reasons). Also, amoplefttype and amoprighttype must match the oprleft and oprright fields of the referenced pg_operator entry.

Table 143: pg_catalog.pg_amop

column	type	references	description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
amopfamily	oid	pg_opfamily.oid	The operator family that this entry is for
amoplefttype	oid	pg_type.oid	Left-hand input data type of operator
amoprighttype	oid	pg_type.oid	Right-hand input data type of operator
amopstrategy	int2		Operator strategy number
amoppurpose	char		Operator purpose, either s for search or o for ordering
amopopr	oid	pg_operator.oid	OID of the operator
amopmethod	oid	pg_am.oid	Index access method for the operator family
amopsortfamily	oid	pg_opfamily.oid	If an ordering operator, the B-tree operator family that this entry sorts according to; zero if a search operator

pg_amproc

The pg_amproc table stores information about support procedures associated with index access method operator classes. There is one row for each support procedure belonging to an operator class.

Table 144: pg_catalog.pg_amproc

column	type	references	description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
amprocfamily	oid	pg_opfamily.oid	The operator family this entry is for

column	type	references	description
amproclefttype	oid	pg_type.oid	Left-hand input data type of associated operator
amprocrighttype	oid	pg_type.oid	Right-hand input data type of associated operator
amprocnum	int2		Support procedure number
amproc	regproc	pg_proc.oid	OID of the procedure

pg_appendonly

The pg_appendonly table contains information about the storage options and other characteristics of append-optimized tables.

Table 145: pg_catalog.pg_appendonly

column	type	references	description
relid	oid		The table object identifier (OID) of the compressed table.
blocksize	integer		Block size used for compression of append-optimized tables. Valid values are 8K - 2M. Default is 32K.
safefswritesize	integer		Minimum size for safe write operations to append-optimized tables in a non-mature file system. Commonly set to a multiple of the extent size of the file system; for example, Linux ext3 is 4096 bytes, so a value of 32768 is commonly used.
compresslevel	smallint		The compression level, with compression ratio increasing from 1 to 19. When quicklz[1] is specified for compresstype, valid values are 1 or 3. With zlib specified, valid values are 1-9. When zstd is specified, valid values are 1-19.

column	type	references	description
majorversion	smallint		The major version number of the pg_ appendonly table.
minorversion	smallint		The minor version number of the pg_ appendonly table.
checksum	boolean		A checksum value that is stored to compare the state of a block of data at compression time and at scan time to ensure data integrity.
compresstype	text		Type of compression used to compress append-optimized tables. Valid values are: • zlib (gzip compression) • zstd (Zstandard compression) • quicklz[1]
columnstore	boolean		1 for column-oriented storage, 0 for row-oriented storage.
segrelid	oid		Table on-disk segment file id.
segidxid	oid		Index on-disk segment file id.
blkdirrelid	oid		Block used for on-disk column-oriented table file.
blkdiridxid	oid		Block used for on-disk column-oriented index file.
visimaprelid	oid		Visibility map for the table.
visimapidxid	oid		B-tree index on the visibility map.

Note: [1]QuickLZ compression is available only in the commercial release of Pivotal Greenplum Database.

pg_attrdef

The pg_attrdef table stores column default values. The main information about columns is stored in *pg_attribute*. Only columns that explicitly specify a default value (when the table is created or the column is added) will have an entry here.

Table 146: pg_catalog.pg_attrdef

column	type	references	description
adrelid	oid	pg_class.oid	The table this column belongs to
adnum	int2	pg_attribute.attnum	The number of the column
adbin	text		The internal representation of the column default value
adsrc	text		A human-readable representation of the default value. This field is historical, and is best not used.

pg_attribute

The pg_attribute table stores information about table columns. There will be exactly one pg_attribute row for every column in every table in the database. (There will also be attribute entries for indexes, and all objects that have pg_class entries.) The term attribute is equivalent to column.

Table 147: pg_catalog.pg_attribute

column	type	references	description
attrelid	oid	pg_class.oid	The table this column belongs to.
attname	name		The column name.
atttypid	oid	pg_type.oid	The data type of this column.
attstattarget	int4		Controls the level of detail of statistics accumulated for this column by ANALYZE. A zero value indicates that no statistics should be collected. A negative value says to use the system default statistics target. The exact meaning of positive values is data type-dependent. For scalar data types, it is both the target number of "most common values" to collect, and the target number of histogram bins to create.

column	type	references	description
attlen	int2		A copy of pg_type.typlen of this column's type.
attnum	int2		The number of the column. Ordinary columns are numbered from 1 up. System columns, such as oid, have (arbitrary) negative numbers.
attndims	int4		Number of dimensions, if the column is an array type; otherwise 0. (Presently, the number of dimensions of an array is not enforced, so any nonzero value effectively means it is an array.)
attcacheoff	int4		Always -1 in storage, but when loaded into a row descriptor in memory this may be updated to cache the offset of the attribute within the row.
atttypmod	int4		Records type-specific data supplied at table creation time (for example, the maximum length of a varchar column). It is passed to type-specific input functions and length coercion functions. The value will generally be -1 for types that do not need it.
attbyval	boolean		A copy of pg_type.typbyval of this column's type.
attstorage	char		Normally a copy of pg_type.typstorage of this column's type. For TOAST-able data types, this can be altered after column creation to control storage policy.

column	type	references	description
attalign	char		A copy of pg_type.typalign of this column's type.
attnotnull	boolean		This represents a not-null constraint. It is possible to change this column to enable or disable the constraint.
atthasdef	boolean		This column has a default value, in which case there will be a corresponding entry in the pg_attrdef catalog that actually defines the value.
attisdropped	boolean		This column has been dropped and is no longer valid. A dropped column is still physically present in the table, but is ignored by the parser and so cannot be accessed via SQL.
attislocal	boolean		This column is defined locally in the relation. Note that a column may be locally defined and inherited simultaneously.
attinhcount	int4		The number of direct ancestors this column has. A column with a nonzero number of ancestors cannot be dropped nor renamed.
attcollation	oid	pg_collation.oid	The defined collation of the column, or zero if the is not of a collatable data type.
attacl	aclitem[]		Column-level access privileges, if any have been granted specifically on this column.
attoptions	text[]		Attribute-level options, as "keyword=value" strings.
attfdwoptions	text[]		Attribute-level foreign data wrapper options, as "keyword=value" strings.

pg_attribute_encoding

The pg_attribute_encoding system catalog table contains column storage information.

Table 148: pg_catalog.pg_attribute_encoding

column	type	modifers	storage	description
attrelid	oid	not null	plain	Foreign key to pg_attribute.attrelid
attnum	smallint	not null	plain	Foreign key to pg_attribute.attnum
attoptions	text [ ]		extended	The options

pg_auth_members

The pg_auth_members system catalog table shows the membership relations between roles. Any non-circular set of relationships is allowed. Because roles are system-wide, pg_auth_members is shared across all databases of a Greenplum Database system.

Table 149: pg_catalog.pg_auth_members

column	type	references	description
roleid	oid	pg_authid.oid	ID of the parent-level (group) role
member	oid	pg_authid.oid	ID of a member role
grantor	oid	pg_authid.oid	ID of the role that granted this membership
admin_option	boolean		True if role member may grant membership to others

pg_authid

The pg_authid table contains information about database authorization identifiers (roles). A role subsumes the concepts of users and groups. A user is a role with the rolcanlogin flag set. Any role (with or without rolcanlogin) may have other roles as members. See *pg_auth_members*.

Since this catalog contains passwords, it must not be publicly readable. *pg_roles* is a publicly readable view on pg_authid that blanks out the password field.

Because user identities are system-wide, pg_authid is shared across all databases in a Greenplum Database system: there is only one copy of pg_authid per system, not one per database.

Table 150: pg_catalog.pg_authid

column	type	references	description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)

column	type	references	description
rolname	name		Role name
rolsuper	boolean		Role has superuser privileges
rolinherit	boolean		Role automatically inherits privileges of roles it is a member of
rolcreaterole	boolean		Role may create more roles
rolcreatedb	boolean		Role may create databases
rolcatupdate	boolean		Role may update system catalogs directly. (Even a superuser may not do this unless this column is true)
rolcanlogin	boolean		Role may log in. That is, this role can be given as the initial session authorization identifier
rolreplication	boolean		Role is a replication role. That is, this role can initiate streaming replication and set/unset the system backup mode using pg_start_backup and pg_stop_backup.
rolconnlimit	int4		For roles that can log in, this sets maximum number of concurrent connections this role can make. -1 means no limit

column	type	references	description
rolpassword	text		Password (possibly encrypted); NULL if none. If the password is encrypted, this column will begin with the string md5 followed by a 32-character hexadecimal MD5 hash. The MD5 hash will be the user's password concatenated to their user name. For example, if user joe has password xyzzy, Greenplum Database will store the md5 hash of xyzzyjoe. Greenplum assumes that a password that does not follow that format is unencrypted.
rolvaliduntil	timestamptz		Password expiry time (only used for password authentication); NULL if no expiration
rolresqueue	oid		Object ID of the associated resource queue ID in *pg_resqueue*
rolcreaterextgpfd	boolean		Privilege to create read external tables with the gpfdist or gpfdists protocol
rolcreaterexhttp	boolean		Privilege to create read external tables with the http protocol
rolcreatewextgpfd	boolean		Privilege to create write external tables with the gpfdist or gpfdists protocol
rolresgroup	oid		Object ID of the associated resource group ID in *pg_resgroup*

pg_available_extension_versions

The pg_available_extension_versions view lists the specific extension versions that are available for installation. The *pg_extension* system catalog table shows the extensions currently installed.

The view is read only.

Table 151: pg_catalog.pg_available_extension_versions

column	type	description
name	name	Extension name.
version	text	Version name.
installed	boolean	True if this version of this extension is currently installed, False otherwise.
superuser	boolean	True if only superusers are allowed to install the extension, False otherwise.
relocatable	boolean	True if extension can be relocated to another schema, False otherwise.
schema	name	Name of the schema that the extension must be installed into, or NULL if partially or fully relocatable.
requires	name[]	Names of prerequisite extensions, or NULL if none
comment	text	Comment string from the extension control file.

pg_available_extensions

The pg_available_extensions view lists the extensions that are available for installation. The *pg_extension* system catalog table shows the extensions currently installed.

The view is read only.

Table 152: pg_catalog.pg_available_extensions

column	type	description
name	name	Extension name.
default_version	text	Name of default version, or NULL if none is specified.
installed_version	text	Currently installed version of the extension, or NULL if not installed.
comment	text	Comment string from the extension control file.

pg_cast

The pg_cast table stores data type conversion paths, both built-in paths and those defined with CREATE CAST.

Note that pg_cast does not represent every type conversion known to the system, only those that cannot be deduced from some generic rule. For example, casting between a domain and its base type is not explicitly represented in pg_cast. Another important exception is that "automatic I/O conversion casts",

those performed using a data type's own I/O functions to convert to or from `text` or other string types, are not explicitly represented in `pg_cast`.

The cast functions listed in `pg_cast` must always take the cast source type as their first argument type, and return the cast destination type as their result type. A cast function can have up to three arguments. The second argument, if present, must be type `integer`; it receives the type modifier associated with the destination type, or `-1` if there is none. The third argument, if present, must be type `boolean`; it receives `true` if the cast is an explicit cast, `false` otherwise.

It is legitimate to create a `pg_cast` entry in which the source and target types are the same, if the associated function takes more than one argument. Such entries represent 'length coercion functions' that coerce values of the type to be legal for a particular type modifier value.

When a `pg_cast` entry has different source and target types and a function that takes more than one argument, the entry converts from one type to another and applies a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two steps, one to convert between data types and a second to apply the modifier.

Table 153: pg_catalog.pg_cast

column	type	references	description
`castsource`	oid	pg_type.oid	OID of the source data type.
`casttarget`	oid	pg_type.oid	OID of the target data type.
`castfunc`	oid	pg_proc.oid	The OID of the function to use to perform this cast. Zero is stored if the cast method does not require a function.
`castcontext`	char		Indicates what contexts the cast may be invoked in. `e` means only as an explicit cast (using CAST or `::` syntax). `a` means implicitly in assignment to a target column, as well as explicitly. `i` means implicitly in expressions, as well as the other cases.
`castmethod`	char		Indicates how the cast is performed: `f` - The function identified in the `castfunc` field is used. `i` - The input/output functions are used. b - The types are binary-coercible, and no conversion is required.

pg_class

The system catalog table `pg_class` catalogs tables and most everything else that has columns or is otherwise similar to a table (also known as *relations*). This includes indexes (see also *pg_index*), sequences, views, composite types, and TOAST tables. Not all columns are meaningful for all relation types.

Table 154: pg_catalog.pg_class

column	type	references	description
`relname`	name		Name of the table, index, view, etc.
`relnamespace`	oid	pg_namespace.oid	The OID of the namespace (schema) that contains this relation
`reltype`	oid	pg_type.oid	The OID of the data type that corresponds to this table's row type, if any (zero for indexes, which have no pg_type entry)
`reloftype`	oid	pg_type.oid	The OID of an entry in `pg_type` for an underlying composite type.
`relowner`	oid	pg_authid.oid	Owner of the relation
`relam`	oid	pg_am.oid	If this is an index, the access method used (B-tree, Bitmap, hash, etc.)
`relfilenode`	oid		Name of the on-disk file of this relation; `0` if none.
`reltablespace`	oid	pg_tablespace.oid	The tablespace in which this relation is stored. If zero, the database's default tablespace is implied. (Not meaningful if the relation has no on-disk file.)
`relpages`	int4		Size of the on-disk representation of this table in pages (of 32K each). This is only an estimate used by the planner. It is updated by `VACUUM`, `ANALYZE`, and a few DDL commands.

column	type	references	description
reltuples	float4		Number of rows in the table. This is only an estimate used by the planner. It is updated by VACUUM, ANALYZE, and a few DDL commands.
relallvisible	int32		Number of all-visible blocks (this value may not be up-to-date).
reltoastrelid	oid	pg_class.oid	OID of the TOAST table associated with this table, 0 if none. The TOAST table stores large attributes "out of line" in a secondary table.
relhasindex	boolean		True if this is a table and it has (or recently had) any indexes. This is set by CREATE INDEX, but not cleared immediately by DROP INDEX. VACUUM will clear if it finds the table has no indexes.
relisshared	boolean		True if this table is shared across all databases in the system. Only certain system catalog tables are shared.
relpersistence	char		The type of object persistence: p = heap or append-optimized table, u = unlogged temporary table, t = temporary table.

column	type	references	description
relkind	char		The type of object r = heap or append-optimized table, i = index, S = sequence, t = TOAST value, v = view, c = composite type, f = foreign table, u = uncatalogued temporary heap table, o = internal append-optimized segment files and EOFs, b = append-only block directory, M = append-only visibility map
relstorage	char		The storage mode of a table a= append-optimized, c= column-oriented, h = heap, v = virtual, x= external table.
relnatts	int2		Number of user columns in the relation (system columns not counted). There must be this many corresponding entries in pg_attribute.
relchecks	int2		Number of check constraints on the table.
relhasoids	boolean		True if an OID is generated for each row of the relation.
relhaspkey	boolean		True if the table has (or once had) a primary key.
relhasrules	boolean		True if table has rules.
relhastriggers	boolean		True if table has (or once had) triggers.
relhassubclass	boolean		True if table has (or once had) any inheritance children.
relispopulated	boolean		True if relation is populated (this is true for all relations other than some materialized views).

column	type	references	description
relreplident	char		Columns used to form “replica identity” for rows: d = default (primary key, if any), n = nothing, f = all columns i = index with indisreplident set, or default
relfrozenxid	xid		All transaction IDs before this one have been replaced with a permanent (frozen) transaction ID in this table. This is used to track whether the table needs to be vacuumed in order to prevent transaction ID wraparound or to allow pg_clog to be shrunk. The value is 0 (InvalidTransactionId) if the relation is not a table or if the table does not require vacuuming to prevent transaction ID wraparound. The table still might require vacuuming to reclaim disk space.
relminmxid	xid		All multixact IDs before this one have been replaced by a transaction ID in this table. This is used to track whether the table needs to be vacuumed in order to prevent multixact ID wraparound or to allow pg_multixact to be shrunk. Zero (InvalidMultiXactId) if the relation is not a table.
relacl	aclitem[]		Access privileges assigned by GRANT and REVOKE.
reloptions	text[]		Access-method-specific options, as "keyword=value" strings.

pg_compression

The pg_compression system catalog table describes the compression methods available.

Table 155: pg_catalog.pg_compression

column	type	modifers	storage	description
compname	name	not null	plain	Name of the compression
compconstructor	regproc	not null	plain	Name of compression constructor
compdestructor	regproc	not null	plain	Name of compression destructor
compcompressor	regproc	not null	plain	Name of the compressor
compdecompressor	regproc	not null	plain	Name of the decompressor
compvalidator	regproc	not null	plain	Name of the compression validator
compowner	oid	not null	plain	oid from pg_authid

pg_constraint

The pg_constraint system catalog table stores check, primary key, unique, and foreign key constraints on tables. Column constraints are not treated specially. Every column constraint is equivalent to some table constraint. Not-null constraints are represented in the *pg_attribute* catalog table. Check constraints on domains are stored here, too.

Table 156: pg_catalog.pg_constraint

column	type	references	description
conname	name		Constraint name (not necessarily unique!)
connamespace	oid	pg_namespace.oid	The OID of the namespace (schema) that contains this constraint.
contype	char		c = check constraint, f = foreign key constraint, p = primary key constraint, u = unique constraint.
condeferrable	boolean		Is the constraint deferrable?
condeferred	boolean		Is the constraint deferred by default?

column	type	references	description
conrelid	oid	pg_class.oid	The table this constraint is on; 0 if not a table constraint.
contypid	oid	pg_type.oid	The domain this constraint is on; 0 if not a domain constraint.
confrelid	oid	pg_class.oid	If a foreign key, the referenced table; else 0.
confupdtype	char		Foreign key update action code.
confdeltype	char		Foreign key deletion action code.
confmatchtype	char		Foreign key match type.
conkey	int2[]	pg_attribute.attnum	If a table constraint, list of columns which the constraint constrains.
confkey	int2[]	pg_attribute.attnum	If a foreign key, list of the referenced columns.
conbin	text		If a check constraint, an internal representation of the expression.
consrc	text		If a check constraint, a human-readable representation of the expression. This is not updated when referenced objects change; for example, it won't track renaming of columns. Rather than relying on this field, it is best to use pg_get_constraintdef() to extract the definition of a check constraint.

pg_conversion

The pg_conversion system catalog table describes the available encoding conversion procedures as defined by CREATE CONVERSION.

Table 157: pg_catalog.pg_conversion

column	type	references	description
conname	name		Conversion name (unique within a namespace).

column	type	references	description
connamespace	oid	pg_namespace.oid	The OID of the namespace (schema) that contains this conversion.
conowner	oid	pg_authid.oid	Owner of the conversion.
conforencoding	int4		Source encoding ID.
contoencoding	int4		Destination encoding ID.
conproc	regproc	pg_proc.oid	Conversion procedure.
condefault	boolean		True if this is the default conversion.

pg_database

The pg_database system catalog table stores information about the available databases. Databases are created with the CREATE DATABASE SQL command. Unlike most system catalogs, pg_database is shared across all databases in the system. There is only one copy of pg_database per system, not one per database.

Table 158: pg_catalog.pg_database

column	type	references	description
datname	name		Database name.
datdba	oid	pg_authid.oid	Owner of the database, usually the user who created it.
encoding	int4		Character encoding for this database. pg_encoding_to_char() can translate this number to the encoding name.
datcollate	name		LC_COLLATE for this database.
datctype	name		LC_CTYPE for this database.
datistemplate	boolean		If true then this database can be used in the TEMPLATE clause of CREATE DATABASE to create a new database as a clone of this one.
datallowconn	boolean		If false then no one can connect to this database. This is used to protect the template0 database from being altered.

column	type	references	description
datconnlimit	int4		Sets the maximum number of concurrent connections that can be made to this database. -1 means no limit.
datlastsysoid	oid		Last system OID in the database.
datfrozenxid	xid		All transaction IDs (XIDs) before this one have been replaced with a permanent (frozen) transaction ID in this database. This is used to track whether the database needs to be vacuumed in order to prevent transaction ID wraparound or to allow pg_clog to be shrunk. It is the minimum of the per-table *pg_class.relfrozenxid* values.
datminmxid	xid		A *Multixact ID* is used to support row locking by multiple transactions. All multixact IDs before this one have been replaced with a transaction ID in this database. This is used to track whether the database needs to be vacuumed in order to prevent multixact ID wraparound or to allow pg_multixact to be shrunk. It is the minimum of the per-table *pg_class.relminmxid* values.
dattablespace	oid	pg_tablespace.oid	The default tablespace for the database. Within this database, all tables for which *pg_class.reltablespace* is zero will be stored in this tablespace. All non-shared system catalogs will also be there.
datacl	aclitem[]		Database access privileges as given by GRANT and REVOKE.

pg_db_role_setting

The pg_db_role_setting system catalog table records the default values of server configuration settings for each role and database combination.

There is a single copy of pg_db_role_settings per Greenplum Database cluster. This system catalog table is shared across all databases.

You can view the server configuration settings for your Greenplum Database cluster with psql's \drds meta-command.

Table 159: pg_catalog.pg_database

column	type	references	description
setdatabase	oid	pg_database.oid	The database to which the setting is applicable, or zero if the setting is not database-specific.
setrole	oid	pg_authid.oid	The role to which the setting is applicable, or zero if the setting is not role-specific.
setconfig	text[]		Per-database- and per-role-specific defaults for user-settable server configuration parameters.

pg_depend

The pg_depend system catalog table records the dependency relationships between database objects. This information allows DROP commands to find which other objects must be dropped by DROP CASCADE or prevent dropping in the DROP RESTRICT case. See also *pg_shdepend*, which performs a similar function for dependencies involving objects that are shared across a Greenplum system.

In all cases, a pg_depend entry indicates that the referenced object may not be dropped without also dropping the dependent object. However, there are several subflavors identified by deptype:

- **DEPENDENCY_NORMAL (n)** — A normal relationship between separately-created objects. The dependent object may be dropped without affecting the referenced object. The referenced object may only be dropped by specifying CASCADE, in which case the dependent object is dropped, too. Example: a table column has a normal dependency on its data type.
- **DEPENDENCY_AUTO (a)** — The dependent object can be dropped separately from the referenced object, and should be automatically dropped (regardless of RESTRICT or CASCADE mode) if the referenced object is dropped. Example: a named constraint on a table is made autodependent on the table, so that it will go away if the table is dropped.
- **DEPENDENCY_INTERNAL (i)** — The dependent object was created as part of creation of the referenced object, and is really just a part of its internal implementation. A DROP of the dependent object will be disallowed outright (we'll tell the user to issue a DROP against the referenced object, instead). A DROP of the referenced object will be propagated through to drop the dependent object whether CASCADE is specified or not.
- **DEPENDENCY_PIN (p)** — There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by system initialization. The columns for the dependent object contain zeroes.

Table 160: pg_catalog.pg_depend

column	type	references	description
`classid`	oid	pg_class.oid	The OID of the system catalog the dependent object is in.
`objid`	oid	any OID column	The OID of the specific dependent object.
`objsubid`	int4		For a table column, this is the column number. For all other object types, this column is zero.
`refclassid`	oid	pg_class.oid	The OID of the system catalog the referenced object is in.
`refobjid`	oid	any OID column	The OID of the specific referenced object.
`refobjsubid`	int4		For a table column, this is the referenced column number. For all other object types, this column is zero.
`deptype`	char		A code defining the specific semantics of this dependency relationship.

pg_description

The `pg_description` system catalog table stores optional descriptions (comments) for each database object. Descriptions can be manipulated with the `COMMENT` command and viewed with `psql`'s `\d` meta-commands. Descriptions of many built-in system objects are provided in the initial contents of `pg_description`. See also *pg_shdescription*, which performs a similar function for descriptions involving objects that are shared across a Greenplum system.

Table 161: pg_catalog.pg_description

column	type	references	description
`objoid`	oid	any OID column	The OID of the object this description pertains to.
`classoid`	oid	pg_class.oid	The OID of the system catalog this object appears in

column	type	references	description
objsubid	int4		For a comment on a table column, this is the column number. For all other object types, this column is zero.
description	text		Arbitrary text that serves as the description of this object.

pg_enum

The pg_enum table contains entries matching enum types to their associated values and labels. The internal representation of a given enum value is actually the OID of its associated row in pg_enum. The OIDs for a particular enum type are guaranteed to be ordered in the way the type should sort, but there is no guarantee about the ordering of OIDs of unrelated enum types.

Table 162: pg_catalog.pg_enum

Column	Type	References	Description
enumtypid	oid	pgtype.oid	The OID of the pg_type entry owning this enum value
enumlabel	name		The textual label for this enum value

pg_extension

The system catalog table pg_extension stores information about installed extensions.

Table 163: pg_catalog.pg_extension

column	type	references	description
extname	name		Name of the extension.
extowner	oid	pg_authid.oid	Owner of the extension
extnamespace	oid	pg_namespace.oid	Schema containing the extension exported objects.
extrelocatable	boolean		True if the extension can be relocated to another schema.
extversion	text		Version name for the extension.
extconfig	oid[]	pg_class.oid	Array of regclass OIDs for the extension configuration tables, or NULL if none.

column	type	references	description
extcondition	text[]		Array of WHERE-clause filter conditions for the extension configuration tables, or NULL if none.

Unlike most catalogs with a namespace column, extnamespace does not imply that the extension belongs to that schema. Extension names are never schema-qualified. The extnamespace schema indicates the schema that contains most or all of the extension objects. If extrelocatable is true, then this schema must contain all schema-qualifiable objects that belong to the extension.

pg_exttable

The pg_exttable system catalog table is used to track external tables and web tables created by the CREATE EXTERNAL TABLE command.

Table 164: pg_catalog.pg_exttable

column	type	references	description
reloid	oid	pg_class.oid	The OID of this external table.
urilocation	text[]		The URI location(s) of the external table files.
execlocation	text[]		The ON segment locations defined for the external table.
fmttype	char		Format of the external table files: t for text, or c for csv.
fmtopts	text		Formatting options of the external table files, such as the field delimiter, null string, escape character, etc.
options	text[]		The options defined for the external table.
command	text		The OS command to execute when the external table is accessed.
rejectlimit	integer		The per segment reject limit for rows with errors, after which the load will fail.
rejectlimittype	char		Type of reject limit threshold: r for number of rows.
logerrors	bool		1 to log errors, 0 to not.

column	type	references	description
encoding	text		The client encoding.
writable	boolean		0 for readable external tables, 1 for writable external tables.

pg_foreign_data_wrapper

The system catalog table pg_foreign_data_wrapper stores foreign-data wrapper definitions. A foreign-data wrapper is a mechanism by which you access external data residing on foreign servers.

Table 165: pg_catalog.pg_foreign_data_wrapper

column	type	references	description
fdwname	name		Name of the foreign-data wrapper.
fdwowner	oid	pg_authid.oid	Owner of the foreign-data wrapper.
fdwhandler	oid	pg_proc.oid	A reference to a handler function that is responsible for supplying execution routines for the foreign-data wrapper. Zero if no handler is provided.
fdwvalidator	oid	pg_proc.oid	A reference to a validator function that is responsible for checking the validity of the options provided to the foreign-data wrapper. This function also checks the options for foreign servers and user mappings using the foreign-data wrapper. Zero if no validator is provided.
fdwacl	aclitem[]		Access privileges; see *GRANT* and *REVOKE* for details.
fdwoptions	text[]		Foreign-data wrapper-specific options, as "keyword=value" strings.

pg_foreign_server

The system catalog table pg_foreign_server stores foreign server definitions. A foreign server describes a source of external data, such as a remote server. You access a foreign server via a foreign-data wrapper.

Table 166: pg_catalog.pg_foreign_server

column	type	references	description
srvname	name		Name of the foreign server.
srvowner	oid	pg_authid.oid	Owner of the foreign server.
srvfdw	oid	pg_foreign_data_wrapper.oid	OID of the foreign-data wrapper of this foreign server.
srvtype	text		Type of server (optional).
srvversion	text		Version of the server (optional).
srvacl	aclitem[]		Access privileges; see *GRANT* and *REVOKE* for details.
srvoptions	text[]		Foreign server-specific options, as "keyword=value" strings.

pg_foreign_table

The system catalog table pg_foreign_table contains auxiliary information about foreign tables. A foreign table is primarily represented by a pg_class entry, just like a regular table. Its pg_foreign_table entry contains the information that is pertinent only to foreign tables and not any other kind of relation.

Table 167: pg_catalog.pg_foreign_table

column	type	references	description
ftrelid	oid	pg_class.oid	OID of the pg_class entry for this foreign table.
ftserver	oid	pg_foreign_server.oid	OID of the foreign server for this foreign table.
ftoptions	text[]		Foreign table options, as "keyword=value" strings.

pg_index

The pg_index system catalog table contains part of the information about indexes. The rest is mostly in *pg_class*.

Table 168: pg_catalog.pg_index

column	type	references	description
indexrelid	oid	pg_class.oid	The OID of the pg_class entry for this index.

column	type	references	description
indrelid	oid	pg_class.oid	The OID of the pg_class entry for the table this index is for.
indnatts	int2		The number of columns in the index (duplicates pg_class.relnatts).
indisunique	boolean		If true, this is a unique index.
indisprimary	boolean		If true, this index represents the primary key of the table. (indisunique should always be true when this is true.)
indisexclusion	boolean		If true, this index supports an exclusion constraint
indimmediate	boolean		If true, the uniqueness check is enforced immediately on insertion (irrelevant if indisunique is not true)
indisclustered	boolean		If true, the table was last clustered on this index via the CLUSTER command.
indisvalid	boolean		If true, the index is currently valid for queries. False means the index is possibly incomplete: it must still be modified by INSERT/UPDATE operations, but it cannot safely be used for queries.
indcheckxmin	boolean		If true, queries must not use the index until the xmin of this pg_index row is below their TransactionXmin event horizon, because the table may contain broken HOT chains with incompatible rows that they can see

column	type	references	description
indisready	boolean		If true, the index is currently ready for inserts. False means the index must be ignored by INSERT/UPDATE operations
indislive	boolean		If false, the index is in process of being dropped, and should be ignored for all purposes
indisreplident	boolean		If true this index has been chosen as "replica identity" using ALTER TABLE ... REPLICA IDENTITY USING INDEX ...
indkey	int2vector	pg_attribute.attnum	This is an array of indnatts values that indicate which table columns this index indexes. For example a value of 1 3 would mean that the first and the third table columns make up the index key. A zero in this array indicates that the corresponding index attribute is an expression over the table columns, rather than a simple column reference.
indcollation	oidvector		For each column in the index key, this contains the OID of the collation to use for the index.
indclass	oidvector	pg_opclass.oid	For each column in the index key this contains the OID of the operator class to use.
indoption	int2vector		This is an array of indnatts values that store per-column flag bits. The meaning of the bits is defined by the index's access method.

column	type	references	description
indexprs	text		Expression trees (in nodeToString() representation) for index attributes that are not simple column references. This is a list with one element for each zero entry in indkey. NULL if all index attributes are simple references.
indpred	text		Expression tree (in nodeToString() representation) for partial index predicate. NULL if not a partial index.

pg_inherits

The pg_inherits system catalog table records information about table inheritance hierarchies. There is one entry for each direct child table in the database. (Indirect inheritance can be determined by following chains of entries.) In Greenplum Database, inheritance relationships are created by both the INHERITS clause (standalone inheritance) and the PARTITION BY clause (partitioned child table inheritance) of CREATE TABLE.

Table 169: pg_catalog.pg_inherits

column	type	references	description
inhrelid	oid	pg_class.oid	The OID of the child table.
inhparent	oid	pg_class.oid	The OID of the parent table.
inhseqno	int4		If there is more than one direct parent for a child table (multiple inheritance), this number tells the order in which the inherited columns are to be arranged. The count starts at 1.

pg_language

The pg_language system catalog table registers languages in which you can write functions or stored procedures. It is populated by CREATE LANGUAGE.

Table 170: pg_catalog.pg_language

column	type	references	description
lanname	name		Name of the language.

column	type	references	description
lanowner	oid	pg_authid.oid	Owner of the language.
lanispl	boolean		This is false for internal languages (such as SQL) and true for user-defined languages. Currently, pg_dump still uses this to determine which languages need to be dumped, but this may be replaced by a different mechanism in the future.
lanpltrusted	boolean		True if this is a trusted language, which means that it is believed not to grant access to anything outside the normal SQL execution environment. Only superusers may create functions in untrusted languages.
lanplcallfoid	oid	pg_proc.oid	For noninternal languages this references the language handler, which is a special function that is responsible for executing all functions that are written in the particular language.
laninline	oid	pg_proc.oid	This references a function that is responsible for executing inline anonymous code blocks (see the *DO* command). Zero if anonymous blocks are not supported.
lanvalidator	oid	pg_proc.oid	This references a language validator function that is responsible for checking the syntax and validity of new functions when they are created. Zero if no validator is provided.
lanacl	aclitem[]		Access privileges for the language.

pg_largeobject

> **Note:** Greenplum Database does not support the PostgreSQL *large object facility* for streaming user data that is stored in large-object structures.

The `pg_largeobject` system catalog table holds the data making up 'large objects'. A large object is identified by an OID assigned when it is created. Each large object is broken into segments or 'pages' small enough to be conveniently stored as rows in `pg_largeobject`. The amount of data per page is defined to be `LOBLKSIZE` (which is currently `BLCKSZ/4`, or typically 8K).

Each row of `pg_largeobject` holds data for one page of a large object, beginning at byte offset (*pageno* * `LOBLKSIZE`) within the object. The implementation allows sparse storage: pages may be missing, and may be shorter than `LOBLKSIZE` bytes even if they are not the last page of the object. Missing regions within a large object read as zeroes.

Table 171: pg_catalog.pg_largeobject

column	type	references	description
`loid`	oid		Identifier of the large object that includes this page.
`pageno`	int4		Page number of this page within its large object (counting from zero).
`data`	bytea		Actual data stored in the large object. This will never be more than `LOBLKSIZE` bytes and may be less.

pg_listener

The `pg_listener` system catalog table supports the `LISTEN` and `NOTIFY` commands. A listener creates an entry in `pg_listener` for each notification name it is listening for. A notifier scans and updates each matching entry to show that a notification has occurred. The notifier also sends a signal (using the PID recorded in the table) to awaken the listener from sleep.

This table is not currently used in Greenplum Database.

Table 172: pg_catalog.pg_listener

column	type	references	description
`relname`	name		Notify condition name. (The name need not match any actual relation in the database.
`listenerpid`	int4		PID of the server process that created this entry.

column	type	references	description
notification	int4		Zero if no event is pending for this listener. If an event is pending, the PID of the server process that sent the notification.

pg_locks

The pg_locks view provides access to information about the locks held by open transactions within Greenplum Database.

pg_locks contains one row per active lockable object, requested lock mode, and relevant transaction. Thus, the same lockable object may appear many times if multiple transactions are holding or waiting for locks on it. An object with no current locks on it will not appear in the view at all.

There are several distinct types of lockable objects: whole relations (such as tables), individual pages of relations, individual tuples of relations, transaction IDs (both virtual and permanent IDs), and general database objects. Also, the right to extend a relation is represented as a separate lockable object.

Table 173: pg_catalog.pg_locks

column	type	references	description
locktype	text		Type of the lockable object: relation, extend, page, tuple, transactionid, object, userlock, resource queue, or advisory
database	oid	pg_database.oid	OID of the database in which the object exists, zero if the object is a shared object, or NULL if the object is a transaction ID
relation	oid	pg_class.oid	OID of the relation, or NULL if the object is not a relation or part of a relation
page	integer		Page number within the relation, or NULL if the object is not a tuple or relation page
tuple	smallint		Tuple number within the page, or NULL if the object is not a tuple
virtualxid	text		Virtual ID of a transaction, or NULL if the object is not a virtual transaction ID

column	type	references	description
transactionid	xid		ID of a transaction, or NULL if the object is not a transaction ID
classid	oid	pg_class.oid	OID of the system catalog containing the object, or NULL if the object is not a general database object
objid	oid	any OID column	OID of the object within its system catalog, or NULL if the object is not a general database object
objsubid	smallint		For a table column, this is the column number (the classid and objid refer to the table itself). For all other object types, this column is zero. NULL if the object is not a general database object
virtualtransaction	text		Virtual ID of the transaction that is holding or awaiting this lock
pid	integer		Process ID of the server process holding or awaiting this lock. NULL if the lock is held by a prepared transaction
mode	text		Name of the lock mode held or desired by this process
granted	boolean		True if lock is held, false if lock is awaited.
fastpath	boolean		True if lock was taken via fastpath, false if lock is taken via main lock table.
mppsessionid	integer		The id of the client session associated with this lock.
mppiswriter	boolean		Specifies whether the lock is held by a writer process.

column	type	references	description
gp_segment_id	integer		The Greenplum segment id (dbid) where the lock is held.

pg_matviews

The view pg_matviews provides access to useful information about each materialized view in the database.

Table 174: pg_catalog.pg_conversion

column	type	references	description
schemaname	name	pg_namespace. nspname	Name of the schema containing the materialized view
matviewname	name	pg_class.relname	Name of the materialized view
matviewowner	name	pg_authid.rolname	Name of the materialized view's owner
tablespace	name	pg_tablespace.spcname	Name of the tablespace containing the materialized view (NULL if default for the database)
hasindexes	boolean		True if the materialized view has (or recently had) any indexes
ispopulated	boolean		True if the materialized view is currently populated
definition	text		Materialized view definition (a reconstructed SELECT command)

pg_max_external_files

The pg_max_external_files view shows the maximum number of external table files allowed per segment host when using the external table file protocol.

Table 175: pg_catalog.pg_max_external_files

column	type	references	description
hostname	name		The host name used to access a particular segment instance on a segment host.

column	type	references	description
maxfiles	bigint		Number of primary segment instances on the host.

pg_namespace

The pg_namespace system catalog table stores namespaces. A namespace is the structure underlying SQL schemas: each namespace can have a separate collection of relations, types, etc. without name conflicts.

Table 176: pg_catalog.pg_namespace

column	type	references	description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
nspname	name		Name of the namespace
nspowner	oid	pg_authid.oid	Owner of the namespace
nspacl	aclitem[]		Access privileges as given by GRANT and REVOKE

pg_opclass

The pg_opclass system catalog table defines index access method operator classes. Each operator class defines semantics for index columns of a particular data type and a particular index access method. An operator class essentially specifies that a particular operator family is applicable to a particular indexable column data type. The set of operators from the family that are actually usable with the indexed column are those that accept the column's data type as their left-hand input.

An operator class's opcmethod must match the opfmethod of its containing operator family. Also, there must be no more than one pg_opclass row having opcdefault true for any given combination of opcmethod and opcintype.

Table 177: pg_catalog.pg_opclass

column	type	references	description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
opcmethod	oid	pg_am.oid	Index access method operator class is for
opcname	name		Name of this operator class
opcnamespace	oid	pg_namespace.oid	Namespace of this operator class
opcowner	oid	pg_authid.oid	Owner of the operator class

column	type	references	description
opcfamily	oid	pg_opfamily.oid	Operator family containing the operator class
opcintype	oid	pg_type.oid	Data type that the operator class indexes
opcdefault	boolean		True if this operator class is the default for the data type opcintype
opckeytype	oid	pg_type.oid	Type of data stored in index, or zero if same as opcintype

pg_operator

The pg_operator system catalog table stores information about operators, both built-in and those defined by CREATE OPERATOR. Unused column contain zeroes. For example, oprleft is zero for a prefix operator.

Table 178: pg_catalog.pg_operator

column	type	references	description
oid	oid		Row identifier (hidden attribute, must be explicityly selected)
oprname	name		Name of the operator
oprnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this operator
oprowner	oid	pg_authid.oid	Owner of the operator
oprkind	char		b = infix (both), l = prefix (left), r = postfix (right)
oprcanmerge	boolean		This operator supports merge joins
oprcanhash	boolean		This operator supports hash joins
oprleft	oid	pg_type.oid	Type of the left operand
oprright	oid	pg_type.oid	Type of the right operand
oprresult	oid	pg_type.oid	Type of the result
oprcom	oid	pg_operator.oid	Commutator of this operator, if any
oprnegate	oid	pg_operator.oid	Negator of this operator, if any

column	type	references	description
oprcode	regproc	pg_proc.oid	Function that implements this operator
oprrest	regproc	pg_proc.oid	Restriction selectivity estimation function for this operator
oprjoin	regproc	pg_proc.oid	Join selectivity estimation function for this operator

pg_opfamily

The catalog pg_opfamily defines operator families. Each operator family is a collection of operators and associated support routines that implement the semantics specified for a particular index access method. Furthermore, the operators in a family are all compatible in a way that is specified by the access method. The operator family concept allows cross-data-type operators to be used with indexes and to be reasoned about using knowledge of access method semantics.

The majority of the information defining an operator family is not in its pg_opfamily row, but in the associated rows in *pg_amop*, *pg_amproc*, and *pg_opclass*.

Table 179: pg_opfamily

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
opfmethod	oid	pg_am.oid	Index access method operator for this family
opfname	name		Name of this operator family
opfnamespace	oid	pg_namespace.oid	Namespace of this operator family
opfowner	oid	pg_authid.oid	Owner of the operator family

pg_partition

The pg_partition system catalog table is used to track partitioned tables and their inheritance level relationships. Each row of pg_partition represents either the level of a partitioned table in the partition hierarchy, or a subpartition template description. The value of the attribute paristemplate determines what a particular row represents.

Table 180: pg_catalog.pg_partition

column	type	references	description
parrelid	oid	pg_class.oid	The object identifier of the table.
parkind	char		The partition type - R for range or L for list.

column	type	references	description
parlevel	smallint		The partition level of this row: 0 for the top-level parent table, 1 for the first level under the parent table, 2 for the second level, and so on.
paristemplate	boolean		Whether or not this row represents a subpartition template definition (true) or an actual partitioning level (false).
parnatts	smallint		The number of attributes that define this level.
paratts	smallint()		An array of the attribute numbers (as in pg_attribute.attnum) of the attributes that participate in defining this level.
parclass	oidvector	pg_opclass.oid	The operator class identifier(s) of the partition columns.

pg_partition_columns

The pg_partition_columns system view is used to show the partition key columns of a partitioned table.

Table 181: pg_catalog.pg_partition_columns

column	type	references	description
schemaname	name		The name of the schema the partitioned table is in.
tablename	name		The table name of the top-level parent table.
columnname	name		The name of the partition key column.
partitionlevel	smallint		The level of this subpartition in the hierarchy.
position_in_partition_key	integer		For list partitions you can have a composite (multi-column) partition key. This shows the position of the column in a composite key.

pg_partition_encoding

The pg_partition_encoding system catalog table describes the available column compression options for a partition template.

Table 182: pg_catalog.pg_attribute_encoding

column	type	modifers	storage	description
parencoid	oid	not null	plain	
parencattnum	snallint	not null	plain	
parencattoptions	text []		extended	

pg_partition_rule

The pg_partition_rule system catalog table is used to track partitioned tables, their check constraints, and data containment rules. Each row of pg_partition_rule represents either a leaf partition (the bottom level partitions that contain data), or a branch partition (a top or mid-level partition that is used to define the partition hierarchy, but does not contain any data).

Table 183: pg_catalog.pg_partition_rule

column	type	references	description
paroid	oid	pg_partition.oid	Row identifier of the partitioning level (from *pg_partition*) to which this partition belongs. In the case of a branch partition, the corresponding table (identified by pg_partition_rule) is an empty container table. In case of a leaf partition, the table contains the rows for that partition containment rule.
parchildrelid	oid	pg_class.oid	The table identifier of the partition (child table).
parparentrule	oid	pg_partition_rule.paroid	The row identifier of the rule associated with the parent table of this partition.
parname	name		The given name of this partition.
parisdefault	boolean		Whether or not this partition is a default partition.

column	type	references	description
parruleord	smallint		For range partitioned tables, the rank of this partition on this level of the partition hierarchy.
parrangestartincl	boolean		For range partitioned tables, whether or not the starting value is inclusive.
parrangeendincl	boolean		For range partitioned tables, whether or not the ending value is inclusive.
parrangestart	text		For range partitioned tables, the starting value of the range.
parrangeend	text		For range partitioned tables, the ending value of the range.
parrangeevery	text		For range partitioned tables, the interval value of the EVERY clause.
parlistvalues	text		For list partitioned tables, the list of values assigned to this partition.
parreloptions	text		An array describing the storage characteristics of the particular partition.

pg_partition_templates

The pg_partition_templates system view is used to show the subpartitions that were created using a subpartition template.

Table 184: pg_catalog.pg_partition_templates

column	type	references	description
schemaname	name		The name of the schema the partitioned table is in.
tablename	name		The table name of the top-level parent table.

column	type	references	description
partitionname	name		The name of the subpartition (this is the name to use if referring to the partition in an ALTER TABLE command). NULL if the partition was not given a name at create time or generated by an EVERY clause.
partitiontype	text		The type of subpartition (range or list).
partitionlevel	smallint		The level of this subpartition in the hierarchy.
partitionrank	bigint		For range partitions, the rank of the partition compared to other partitions of the same level.
partitionposition	smallint		The rule order position of this subpartition.
partitionlistvalues	text		For list partitions, the list value(s) associated with this subpartition.
partitionrangestart	text		For range partitions, the start value of this subpartition.
partitionstartinclusive	boolean		T if the start value is included in this subpartition. F if it is excluded.
partitionrangeend	text		For range partitions, the end value of this subpartition.
partitionendinclusive	boolean		T if the end value is included in this subpartition. F if it is excluded.
partitioneveryclause	text		The EVERY clause (interval) of this subpartition.
partitionisdefault	boolean		T if this is a default subpartition, otherwise F.

column	type	references	description
partitionboundary	text		The entire partition specification for this subpartition.

pg_partitions

The pg_partitions system view is used to show the structure of a partitioned table.

Table 185: pg_catalog.pg_partitions

column	type	references	description
schemaname	name		The name of the schema the partitioned table is in.
tablename	name		The name of the top-level parent table.
partitiontablename	name		The relation name of the partitioned table (this is the table name to use if accessing the partition directly).
partitionname	name		The name of the partition (this is the name to use if referring to the partition in an ALTER TABLE command). NULL if the partition was not given a name at create time or generated by an EVERY clause.
parentpartitiontablename	name		The relation name of the parent table one level up from this partition.
parentpartitionname	name		The given name of the parent table one level up from this partition.
partitiontype	text		The type of partition (range or list).
partitionlevel	smallint		The level of this partition in the hierarchy.
partitionrank	bigint		For range partitions, the rank of the partition compared to other partitions of the same level.
partitionposition	smallint		The rule order position of this partition.

column	type	references	description
partitionlistvalues	text		For list partitions, the list value(s) associated with this partition.
partitionrangestart	text		For range partitions, the start value of this partition.
partitionstartinclusive	boolean		T if the start value is included in this partition. F if it is excluded.
partitionrangeend	text		For range partitions, the end value of this partition.
partitionendinclusive	boolean		T if the end value is included in this partition. F if it is excluded.
partitioneveryclause	text		The EVERY clause (interval) of this partition.
partitionisdefault	boolean		T if this is a default partition, otherwise F.
partitionboundary	text		The entire partition specification for this partition.

pg_pltemplate

The pg_pltemplate system catalog table stores template information for procedural languages. A template for a language allows the language to be created in a particular database by a simple CREATE LANGUAGE command, with no need to specify implementation details. Unlike most system catalogs, pg_pltemplate is shared across all databases of Greenplum system: there is only one copy of pg_pltemplate per system, not one per database. This allows the information to be accessible in each database as it is needed.

There are not currently any commands that manipulate procedural language templates; to change the built-in information, a superuser must modify the table using ordinary INSERT, DELETE, or UPDATE commands.

Table 186: pg_catalog.pg_pltemplate

column	type	references	description
tmplname	name		Name of the language this template is for
tmpltrusted	boolean		True if language is considered trusted
tmplhandler	text		Name of call handler function
tmplvalidator	text		Name of validator function, or NULL if none

column	type	references	description
tmpllibrary	text		Path of shared library that implements language
tmplacl	aclitem[]		Access privileges for template (not yet implemented).

pg_proc

The pg_proc system catalog table stores information about functions (or procedures), both built-in functions and those defined by CREATE FUNCTION. The table contains data for aggregate and window functions as well as plain functions. If proisagg is true, there should be a matching row in pg_aggregate.

For compiled functions, both built-in and dynamically loaded, prosrc contains the function's C-language name (link symbol). For all other currently-known language types, prosrc contains the function's source text. probin is unused except for dynamically-loaded C functions, for which it gives the name of the shared library file containing the function.

Table 187: pg_catalog.pg_proc

column	type	references	description
oid	oid		Row identifier (hidden attribute; ust be explicitly selected)
proname	name		Name of the function
pronamespace	oid	pg_namespace.oid	The OID of the namespace that contains this function
proowner	oid	pg_authid.oid	Owner of the function
prolang	oid	pg_language.oid	Implementation language or call interface of this function
procost	float4		Estimated execution cost (in cpu_operator_cost units); if proretset is true, identifies the cost per row returned
prorows	float4		Estimated number of result rows (zero if not proretset)
provariadic	oid	pg_type.oid	Data type of the variadic array parameter's elements, or zero if the function does not have a variadic parameter

column	type	references	description
protransform	regproc	pg_proc.oid	Calls to this function can be simplified by this other function
proisagg	boolean		Function is an aggregate function
proiswindow	boolean		Function is a window function
prosecdef	boolean		Function is a security definer (for example, a 'setuid' function)
proleakproof	boolean		The function has no side effects. No information about the arguments is conveyed except via the return value. Any function that might throw an error depending on the values of its arguments is not leak-proof.
proisstrict	boolean		Function returns NULL if any call argument is NULL. In that case the function will not actually be called at all. Functions that are not strict must be prepared to handle NULL inputs.
proretset	boolean		Function returns a set (multiple values of the specified data type)
provolatile	char		Tells whether the function's result depends only on its input arguments, or is affected by outside factors. i = *immutable* (always delivers the same result for the same inputs), s = *stable* (results (for fixed inputs) do not change within a scan), or v = *volatile* (results may change at any time or functions with side-effects).
pronargs	int2		Number of arguments
pronargdefaults	int2		Number of arguments that have default values

column	type	references	description
`prorettype`	oid	pg_type.oid	Data type of the return value
`proargtypes`	oidvector	pg_type.oid	An array with the data types of the function arguments. This includes only input arguments (including `INOUT` and `VARIADIC` arguments), and thus represents the call signature of the function.
`proallargtypes`	oid[]	pg_type.oid	An array with the data types of the function arguments. This includes all arguments (including `OUT` and `INOUT` arguments); however, if all of the arguments are `IN` arguments, this field will be null. Note that subscripting is 1-based, whereas for historical reasons `proargtypes` is subscripted from 0.
`proargmodes`	char[]		An array with the modes of the function arguments: `i` = `IN`, `o` = `OUT` , `b` = `INOUT`, `v` = `VARIADIC`. If all the arguments are IN arguments, this field will be null. Note that subscripts correspond to positions of `proallargtypes`, not `proargtypes`.
`proargnames`	text[]		An array with the names of the function arguments. Arguments without a name are set to empty strings in the array. If none of the arguments have a name, this field will be null. Note that subscripts correspond to positions of `proallargtypes` not `proargtypes`.

column	type	references	description
proargdefaults	pg_node_tree		Expression trees (in nodeToString() representation) for default argument values. This is a list with pronargdefaults elements, corresponding to the last *N* input arguments (i.e., the last *N* proargtypes positions). If none of the arguments have defaults, this field will be null.
prosrc	text		This tells the function handler how to invoke the function. It might be the actual source code of the function for interpreted languages, a link symbol, a file name, or just about anything else, depending on the implementation language/call convention.
probin	text		Additional information about how to invoke the function. Again, the interpretation is language-specific.
proconfig	text[]		Function's local settings for run-time configuration variables.
proacl	aclitem[]		Access privileges for the function as given by GRANT/REVOKE
prodataaccess	char		Provides a hint regarding the type SQL statements that are included in the function: n - does not contain SQL, c - contains SQL, r - contains SQL that reads data, m - contains SQL that modifies data

column	type	references	description
proexeclocation	char		Where the function executes when it is invoked: m - master only, a - any segment instance, s - all segment instances.

pg_resgroup

Note: The pg_resgroup system catalog table is valid only when resource group-based resource management is active.

The pg_resgroup system catalog table contains information about Greenplum Database resource groups, which are used for managing concurrent statements, CPU, and memory resources. This table, defined in the pg_global tablespace, is globally shared across all databases in the system.

Table 188: pg_catalog.pg_resgroup

column	type	references	description
rsgname	name		The name of the resource group.
parent	oid		Unused; reserved for future use.

pg_resgroupcapability

Note: The pg_resgroupcapability system catalog table is valid only when resource group-based resource management is active.

The pg_resgroupcapability system catalog table contains information about the capabilities and limits of defined Greenplum Database resource groups. You can join this table to the *pg_resgroup* table by resource group object ID.

The pg_resgroupcapability table, defined in the pg_global tablespace, is globally shared across all databases in the system.

Table 189: pg_catalog.pg_resgroupcapability

column	type	references	description
resgroupid	oid	pg_resgroup.oid	The object ID of the associated resource group.

column	type	references	description
reslimittype	smallint		The resource group limit type: 0 - Unknown 1 - Concurrency 2 - CPU 3 - Memory 4 - Memory shared quota 5 - Memory spill ratio 6 - Memory auditor 7 - CPU set
value	opaque type		The specific value set for the resource limit referenced in this record. This value has the fixed type text, and will be converted to a different data type depending upon the limit referenced.

pg_resourcetype

The pg_resourcetype system catalog table contains information about the extended attributes that can be assigned to Greenplum Database resource queues. Each row details an attribute and inherent qualities such as its default setting, whether it is required, and the value to disable it (when allowed).

This table is populated only on the master. This table is defined in the pg_global tablespace, meaning it is globally shared across all databases in the system.

Table 190: pg_catalog.pg_resourcetype

column	type	references	description
restypid	smallint		The resource type ID.
resname	name		The name of the resource type.
resrequired	boolean		Whether the resource type is required for a valid resource queue.
reshasdefault	boolean		Whether the resource type has a default value. When true, the default value is specified in reshasdefaultsetting.

column	type	references	description
rescandisable	boolean		Whether the type can be removed or disabled. When true, the default value is specified in resdisabledsetting.
resdefaultsetting	text		Default setting for the resource type, when applicable.
resdisabledsetting	text		The value that disables this resource type (when allowed).

pg_resqueue

Note: The pg_resqueue system catalog table is valid only when resource queue-based resource management is active.

The pg_resqueue system catalog table contains information about Greenplum Database resource queues, which are used for the resource management feature. This table is populated only on the master. This table is defined in the pg_global tablespace, meaning it is globally shared across all databases in the system.

Table 191: pg_catalog.pg_resqueue

column	type	references	description
rsqname	name		The name of the resource queue.
rsqcountlimit	real		The active query threshold of the resource queue.
rsqcostlimit	real		The query cost threshold of the resource queue.
rsqovercommit	boolean		Allows queries that exceed the cost threshold to run when the system is idle.
rsqignorecostlimit	real		The query cost limit of what is considered a 'small query'. Queries with a cost under this limit will not be queued and run immediately.

pg_resqueue_attributes

Note: The pg_resqueue_attributes view is valid only when resource queue-based resource management is active.

The pg_resqueue_attributes view allows administrators to see the attributes set for a resource queue, such as its active statement limit, query cost limits, and priority.

Table 192: pg_catalog.pg_resqueue_attributes

column	type	references	description
rsqname	name	pg_resqueue.rsqname	The name of the resource queue.
resname	text		The name of the resource queue attribute.
resetting	text		The current value of a resource queue attribute.
restypid	integer		System assigned resource type id.

pg_resqueuecapability

> **Note:** The pg_resqueuecapability system catalog table is valid only when resource queue-based resource management is active.

The pg_resqueuecapability system catalog table contains information about the extended attributes, or capabilities, of existing Greenplum Database resource queues. Only resource queues that have been assigned an extended capability, such as a priority setting, are recorded in this table. This table is joined to the *pg_resqueue* table by resource queue object ID, and to the *pg_resourcetype* table by resource type ID (restypid).

This table is populated only on the master. This table is defined in the pg_global tablespace, meaning it is globally shared across all databases in the system.

Table 193: pg_catalog.pg_resqueuecapability

column	type	references	description
rsqueueid	oid	pg_resqueue.oid	The object ID of the associated resource queue.
restypid	smallint	pg_resourcetype. restypid	The resource type, derived from the *pg_ resqueuecapability* system table.
resetting	opaque type		The specific value set for the capability referenced in this record. Depending on the actual resource type, this value may have different data types.

pg_rewrite

The pg_rewrite system catalog table stores rewrite rules for tables and views. pg_class.relhasrules must be true if a table has any rules in this catalog.

Table 194: pg_catalog.pg_rewrite

column	type	references	description
rulename	name		Rule name.
ev_class	oid	pg_class.oid	The table this rule is for.
ev_type	char		Event type that the rule is for: 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE
ev_enabled	char		Controls in which session replication role mode the rule fires. Always O, rule fires in origin mode.
is_instead	bool		True if the rule is an `INSTEAD` rule
ev_qual	pg_node_tree		Expression tree (in the form of a `nodeToString()` representation) for the rule's qualifying condition
ev_action	pg_node_tree		Query tree (in the form of a `nodeToString()` representation) for the rule's action

pg_roles

The view `pg_roles` provides access to information about database roles. This is simply a publicly readable view of *pg_authid* that blanks out the password field. This view explicitly exposes the OID column of the underlying table, since that is needed to do joins to other catalogs.

Table 195: pg_catalog.pg_roles

column	type	references	description
`rolname`	name		Role name
`rolsuper`	bool		Role has superuser privileges
`rolinherit`	bool		Role automatically inherits privileges of roles it is a member of
`rolcreaterole`	bool		Role may create more roles
`rolcreatedb`	bool		Role may create databases

column	type	references	description
rolcatupdate	bool		Role may update system catalogs directly. (Even a superuser may not do this unless this column is true.)
rolcanlogin	bool		Role may log in. That is, this role can be given as the initial session authorization identifier
rolconnlimit	int4		For roles that can log in, this sets maximum number of concurrent connections this role can make. -1 means no limit
rolpassword	text		Not the password (always reads as ********)
rolvaliduntil	timestamptz		Password expiry time (only used for password authentication); NULL if no expiration
rolconfig	text[]		Role-specific defaults for run-time configuration variables
rolresqueue	oid	pg_resqueue.oid	Object ID of the resource queue this role is assigned to.
oid	oid	pg_authid.oid	Object ID of role
rolcreaterextgpfd	bool		Role may create readable external tables that use the gpfdist protocol.
rolcreaterexthttp	bool		Role may create readable external tables that use the http protocol.
rolcreatewextgpfd	bool		Role may create writable external tables that use the gpfdist protocol.
rolresgroup	oid	pg_resgroup.oid	Object ID of the resource group to which this role is assigned.

pg_rules

The view pg_rules provides access to useful information about query rewrite rules.

The pg_rules view excludes the ON SELECT rules of views and materialized views; those can be seen in pg_views and pg_matviews.

Table 196: pg_catalog.pg_rules

column	type	references	description
schemaname	name	pg_namespace.nspname	Name of schema containing table
tablename	name	pg_class.relname	Name of table the rule is for
rulename	name	pg_rewrite.rulename	Name of rule
definition	text		Rule definition (a reconstructed creation command)

pg_shdepend

The pg_shdepend system catalog table records the dependency relationships between database objects and shared objects, such as roles. This information allows Greenplum Database to ensure that those objects are unreferenced before attempting to delete them. See also *pg_depend*, which performs a similar function for dependencies involving objects within a single database. Unlike most system catalogs, pg_shdepend is shared across all databases of Greenplum system: there is only one copy of pg_shdepend per system, not one per database.

In all cases, a pg_shdepend entry indicates that the referenced object may not be dropped without also dropping the dependent object. However, there are several subflavors identified by deptype:

- **SHARED_DEPENDENCY_OWNER (o)** — The referenced object (which must be a role) is the owner of the dependent object.
- **SHARED_DEPENDENCY_ACL (a)** — The referenced object (which must be a role) is mentioned in the ACL (access control list) of the dependent object.
- **SHARED_DEPENDENCY_PIN (p)** — There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by system initialization. The columns for the dependent object contain zeroes.

Table 197: pg_catalog.pg_shdepend

column	type	references	description
dbid	oid	pg_database.oid	The OID of the database the dependent object is in, or zero for a shared object.
classid	oid	pg_class.oid	The OID of the system catalog the dependent object is in.
objid	oid	any OID column	The OID of the specific dependent object.
objsubid	int4		For a table column, this is the column number. For all other object types, this column is zero.

column	type	references	description
refclassid	oid	pg_class.oid	The OID of the system catalog the referenced object is in (must be a shared catalog).
refobjid	oid	any OID column	The OID of the specific referenced object.
refobjsubid	int4		For a table column, this is the referenced column number. For all other object types, this column is zero.
deptype	char		A code defining the specific semantics of this dependency relationship.

pg_shdescription

The pg_shdescription system catalog table stores optional descriptions (comments) for shared database objects. Descriptions can be manipulated with the COMMENT command and viewed with psql's \d meta-commands. See also *pg_description*, which performs a similar function for descriptions involving objects within a single database. Unlike most system catalogs, pg_shdescription is shared across all databases of a Greenplum system: there is only one copy of pg_shdescription per system, not one per database.

Table 198: pg_catalog.pg_shdescription

column	type	references	description
objoid	oid	any OID column	The OID of the object this description pertains to.
classoid	oid	pg_class.oid	The OID of the system catalog this object appears in
description	text		Arbitrary text that serves as the description of this object.

pg_stat_activity

The view pg_stat_activity shows one row per server process with details about the associated user session and query. The columns that report data on the current query are available unless the parameter stats_command_string has been turned off. Furthermore, these columns are only visible if the user examining the view is a superuser or the same as the user owning the process being reported on.

The maximum length of the query text string stored in the column query can be controlled with the server configuration parameter track_activity_query_size.

Table 199: pg_catalog.pg_stat_activity

column	type	references	description
datid	oid	pg_database.oid	Database OID
datname	name		Database name
pid	integer		Process ID of this backend
sess_id	integer		Session ID
usesysid	oid	pg_authid.oid	OID of the user logged into this backend
usename	name		Name of the user logged into this backend
application_name	text		Name of the application that is connected to this backend
client_addr	inet		IP address of the client connected to this backend. If this field is null, it indicates either that the client is connected via a Unix socket on the server machine or that this is an internal process such as autovacuum.
client_hostname	text		Host name of the connected client, as reported by a reverse DNS lookup of client_addr. This field will only be non-null for IP connections, and only when log_hostname is enabled.
client_port	integer		TCP port number that the client is using for communication with this backend, or -1 if a Unix socket is used
backend_start	timestamptz		Time backend process was started
xact_start	timestamptz		Transaction start time
query_start	timestamptz		Time query began execution
state_change	timestampz		Time when the state was last changed

column	type	references	description
`waiting`	boolean		True if waiting on a lock, false if not waiting
`state`	text		Current overall state of this backend. Possible values are: • `active`: The backend is executing a query. • `idle`: The backend is waiting for a new client command. • `idle in transaction`: The backend is in a transaction, but is not currently executing a query. • `idle in transaction (aborted)`: This state is similar to idle in transaction, except one of the statements in the transaction caused an error. • `fastpath function call`: The backend is executing a fast-path function. • `disabled`: This state is reported if `track_activities` is disabled in this backend.
`query`	text		Text of this backend's most recent query. If `state` is active this field shows the currently executing query. In all other states, it shows the last query that was executed.
`waiting_reason`	text		Reason the server process is waiting. The value can be: lock, replication, or resgroup

column	type	references	description
rsgid	oid	pg_resgroup.oid	Resource group OID or 0. See *Note*.
rsgname	text	pg_resgroup.rsgname	Resource group name or unknown. See *Note*.
rsgqueueduration	interval		For a queued query, the total time the query has been queued.

Note: When resource groups are enabled. Only query dispatcher (QD) processes will have a rsgid and rsgname. Other server processes such as a query executer (QE) process or session connection processes will have a rsgid value of 0 and a rsgname value of unknown. QE processes are managed by the same resource group as the dispatching QD process.

pg_stat_all_indexes

The pg_stat_all_indexes view shows one row for each index in the current database that displays statistics about accesses to that specific index.

The pg_stat_user_indexes and pg_stat_sys_indexes views contain the same information, but filtered to only show user and system indexes respectively.

In Greenplum Database 6, the pg_stat_*_indexes views display access statistics for indexes only from the master instance. Access statistics from segment instances are ignored. You can create views that display usage statistics that combine statistics from the master and the segment instances, see *Index Access Statistics from the Master and Segment Instances*.

Table 200: pg_catalog.pg_stat_all_indexes View

Column	Type	Description
relid	oid	OID of the table for this index
indexrelid	oid	OID of this index
schemaname	name	Name of the schema this index is in
relname	name	Name of the table for this index
indexrelname	name	Name of this index
idx_scan	bigint	Total number of index scans initiated on this index from all segment instances
idx_tup_read	bigint	Number of index entries returned by scans on this index
idx_tup_fetch	bigint	Number of live table rows fetched by simple index scans using this index

Index Access Statistics from the Master and Segment Instances

To display index access statistics that combine statistics from the master and the segment instances you can create these views. A user requires `SELECT` privilege on the views to use them.

```
-- Create these index access statistics views
--   pg_stat_all_indexes_gpdb6
--   pg_stat_sys_indexes_gpdb6
--   pg_stat_user_indexes_gpdb6

CREATE VIEW pg_stat_all_indexes_gpdb6 AS
SELECT
    s.relid,
    s.indexrelid,
    s.schemaname,
    s.relname,
    s.indexrelname,
    m.idx_scan,
    m.idx_tup_read,
    m.idx_tup_fetch
FROM
    (SELECT
         relid,
         indexrelid,
         schemaname,
         relname,
         indexrelname,
         sum(idx_scan) as idx_scan,
         sum(idx_tup_read) as idx_tup_read,
         sum(idx_tup_fetch) as idx_tup_fetch
     FROM gp_dist_random('pg_stat_all_indexes')
     WHERE relid >= 16384
     GROUP BY relid, indexrelid, schemaname, relname, indexrelname
     UNION ALL
     SELECT *
     FROM pg_stat_all_indexes
     WHERE relid < 16384) m, pg_stat_all_indexes s
WHERE m.relid = s.relid;


CREATE VIEW pg_stat_sys_indexes_gpdb6 AS
    SELECT * FROM pg_stat_all_indexes_gpdb6
    WHERE schemaname IN ('pg_catalog', 'information_schema') OR
          schemaname ~ '^pg_toast';


CREATE VIEW pg_stat_user_indexes_gpdb6 AS
    SELECT * FROM pg_stat_all_indexes_gpdb6
    WHERE schemaname NOT IN ('pg_catalog', 'information_schema') AND
          schemaname !~ '^pg_toast';
```

pg_stat_all_tables

The `pg_stat_all_tables` view shows one row for each table in the current database (including TOAST tables) to display statistics about accesses to that specific table.

The `pg_stat_user_tables` and `pg_stat_sys_tables` views contain the same information, but filtered to only show user and system tables respectively.

In Greenplum Database 6, the `pg_stat_*_tables` views display access statistics for tables only from the master instance. Access statistics from segment instances are ignored. You can create views that display usage statistics, see *Table Access Statistics from the Master and Segment Instances*.

Table 201: pg_catalog.pg_stat_all_table View

Column	Type	Description
relid	oid	OID of a table
schemaname	name	Name of the schema that this table is in
relname	name	Name of this table
seq_scan	bigint	Total number of sequential scans initiated on this table from all segment instances
seq_tup_read	bigint	Number of live rows fetched by sequential scans
idx_scan	bigint	Total number of index scans initiated on this index from all segment instances
idx_tup_fetch	bigint	Number of live rows fetched by index scans
n_tup_ins	bigint	Number of rows inserted
n_tup_upd	bigint	Number of rows updated (includes HOT updated rows)
n_tup_del	bigint	Number of rows deleted
n_tup_hot_upd	bigint	Number of rows HOT updated (i.e., with no separate index update required)
n_live_tup	bigint	Estimated number of live rows
n_dead_tup	bigint	Estimated number of dead rows
n_mod_since_analyze	bigint	Estimated number of rows modified since this table was last analyzed
last_vacuum	timestamp with time zone	Last time this table was manually vacuumed (not counting VACUUM FULL)
last_autovacuum	timestamp with time zone	Last time this table was vacuumed by the autovacuum daemon[1]
last_analyze	timestamp with time zone	Last time this table was manually analyzed
last_autoanalyze	timestamp with time zone	Last time this table was analyzed by the autovacuum daemon[1]
vacuum_count	bigint	Number of times this table has been manually vacuumed (not counting VACUUM FULL)

Column	Type	Description
autovacuum_count	bigint	Number of times this table has been vacuumed by the autovacuum daemon[1]
analyze_count	bigint	Number of times this table has been manually analyzed
autoanalyze_count	bigint	Number of times this table has been analyzed by the autovacuum daemon [1]

Note: [1] In Greenplum Database, the autovacuum daemon is disabled and not supported for user defined databases.

Table Access Statistics from the Master and Segment Instances

To display table access statistics that combine statistics from the master and the segment instances you can create these views. A user requires SELECT privilege on the views to use them.

```
-- Create these table access statistics views
--   pg_stat_all_tables_gpdb6
--   pg_stat_sys_tables_gpdb6
--   pg_stat_user_tables_gpdb6

CREATE VIEW pg_stat_all_tables_gpdb6 AS
SELECT
    s.relid,
    s.schemaname,
    s.relname,
    m.seq_scan,
    m.seq_tup_read,
    m.idx_scan,
    m.idx_tup_fetch,
    m.n_tup_ins,
    m.n_tup_upd,
    m.n_tup_del,
    m.n_tup_hot_upd,
    m.n_live_tup,
    m.n_dead_tup,
    s.n_mod_since_analyze,
    s.last_vacuum,
    s.last_autovacuum,
    s.last_analyze,
    s.last_autoanalyze,
    s.vacuum_count,
    s.autovacuum_count,
    s.analyze_count,
    s.autoanalyze_count
FROM
    (SELECT
         relid,
         schemaname,
         relname,
         sum(seq_scan) as seq_scan,
         sum(seq_tup_read) as seq_tup_read,
         sum(idx_scan) as idx_scan,
         sum(idx_tup_fetch) as idx_tup_fetch,
         sum(n_tup_ins) as n_tup_ins,
         sum(n_tup_upd) as n_tup_upd,
         sum(n_tup_del) as n_tup_del,
```

```
          sum(n_tup_hot_upd) as n_tup_hot_upd,
          sum(n_live_tup) as n_live_tup,
          sum(n_dead_tup) as n_dead_tup,
          max(n_mod_since_analyze) as n_mod_since_analyze,
          max(last_vacuum) as last_vacuum,
          max(last_autovacuum) as last_autovacuum,
          max(last_analyze) as last_analyze,
          max(last_autoanalyze) as last_autoanalyze,
          max(vacuum_count) as vacuum_count,
          max(autovacuum_count) as autovacuum_count,
          max(analyze_count) as analyze_count,
          max(autoanalyze_count) as autoanalyze_count
      FROM gp_dist_random('pg_stat_all_tables')
      WHERE relid >= 16384
      GROUP BY relid, schemaname, relname
      UNION ALL
      SELECT *
      FROM pg_stat_all_tables
      WHERE relid < 16384) m, pg_stat_all_tables s
 WHERE m.relid = s.relid;


CREATE VIEW pg_stat_sys_tables_gpdb6 AS
    SELECT * FROM pg_stat_all_tables_gpdb6
    WHERE schemaname IN ('pg_catalog', 'information_schema') OR
          schemaname ~ '^pg_toast';


CREATE VIEW pg_stat_user_tables_gpdb6 AS
    SELECT * FROM pg_stat_all_tables_gpdb6
    WHERE schemaname NOT IN ('pg_catalog', 'information_schema') AND
          schemaname !~ '^pg_toast';
```

pg_stat_last_operation

The pg_stat_last_operation table contains metadata tracking information about database objects (tables, views, etc.).

Table 202: pg_catalog.pg_stat_last_operation

column	type	references	description
classid	oid	pg_class.oid	OID of the system catalog containing the object.
objid	oid	any OID column	OID of the object within its system catalog.
staactionname	name		The action that was taken on the object.
stasysid	oid	pg_authid.oid	A foreign key to pg_ authid.oid.
stausename	name		The name of the role that performed the operation on this object.

column	type	references	description
stasubtype	text		The type of object operated on or the subclass of operation performed.
statime	timestamp with timezone		The timestamp of the operation. This is the same timestamp that is written to the Greenplum Database server log files in case you need to look up more detailed information about the operation in the logs.

The pg_stat_last_operation table contains metadata tracking information about operations on database objects. This information includes the object id, DDL action, user, type of object, and operation timestamp. Greenplum Database updates this table when a database object is created, altered, truncated, vacuumed, analyzed, or partitioned, and when privileges are granted to an object.

If you want to track the operations performed on a specific object, use the objid value. Because the stasubtype value can identify either the type of object operated on or the subclass of operation performed, it is not a suitable parameter when querying the pg_stat_last_operation table.

The following example creates and replaces a view, and then shows how to use objid as a query parameter on the pg_stat_last_operation table.

```
testdb=# CREATE VIEW trial AS SELECT * FROM gp_segment_configuration;
CREATE VIEW
testdb=# CREATE OR REPLACE VIEW trial AS SELECT * FROM
 gp_segment_configuration;
CREATE VIEW
testdb=# SELECT * FROM pg_stat_last_operation WHERE
 objid='trial'::regclass::oid;
 classid | objid | staactionname | stasysid | stausename | stasubtype |
       statime
---------+-------+---------------+----------+------------+------------
+-------------------------------
  1259  | 24735 | CREATE         |       10 | gpadmin    | VIEW       |
 2020-04-07 16:44:28.808811+00
  1259  | 24735 | ALTER          |       10 | gpadmin    | SET        |
 2020-04-07 16:44:38.110615+00
(2 rows)
```

Notice that the pg_stat_last_operation table entry for the view REPLACE operation specifies the ALTER action (staactionname) and the SET subtype (stasubtype).

pg_stat_last_shoperation

The pg_stat_last_shoperation table contains metadata tracking information about global objects (roles, tablespaces, etc.).

Table 203: pg_catalog.pg_stat_last_shoperation

column	type	references	description
classid	oid	pg_class.oid	OID of the system catalog containing the object.
objid	oid	any OID column	OID of the object within its system catalog.
staactionname	name		The action that was taken on the object.
stasysid	oid		
stausename	name		The name of the role that performed the operation on this object.
stasubtype	text		The type of object operated on or the subclass of operation performed.
statime	timestamp with timezone		The timestamp of the operation. This is the same timestamp that is written to the Greenplum Database server log files in case you need to look up more detailed information about the operation in the logs.

pg_stat_operations

The view pg_stat_operations shows details about the last operation performed on a database object (such as a table, index, view or database) or a global object (such as a role).

Table 204: pg_catalog.pg_stat_operations

column	type	references	description
classname	text		The name of the system table in the pg_catalog schema where the record about this object is stored (pg_class=relations, pg_database=databases, pg_namespace=schemas, pg_authid=roles)
objname	name		The name of the object.
objid	oid		The OID of the object.

column	type	references	description
schemaname	name		The name of the schema where the object resides.
usestatus	text		The status of the role who performed the last operation on the object (CURRENT=a currently active role in the system, DROPPED=a role that no longer exists in the system, CHANGED=a role name that exists in the system, but has changed since the last operation was performed).
username	name		The name of the role that performed the operation on this object.
actionname	name		The action that was taken on the object.
subtype	text		The type of object operated on or the subclass of operation performed.
statime	timestamptz		The timestamp of the operation. This is the same timestamp that is written to the Greenplum Database server log files in case you need to look up more detailed information about the operation in the logs.

pg_stat_partition_operations

The pg_stat_partition_operations view shows details about the last operation performed on a partitioned table.

Table 205: pg_catalog.pg_stat_partition_operations

column	type	references	description
classname	text		The name of the system table in the pg_catalog schema where the record about this object is stored (always pg_class for tables and partitions).
objname	name		The name of the object.

column	type	references	description
objid	oid		The OID of the object.
schemaname	name		The name of the schema where the object resides.
usestatus	text		The status of the role who performed the last operation on the object (CURRENT=a currently active role in the system, DROPPED=a role that no longer exists in the system, CHANGED=a role name that exists in the system, but its definition has changed since the last operation was performed).
usename	name		The name of the role that performed the operation on this object.
actionname	name		The action that was taken on the object.
subtype	text		The type of object operated on or the subclass of operation performed.
statime	timestamptz		The timestamp of the operation. This is the same timestamp that is written to the Greenplum Database server log files in case you need to look up more detailed information about the operation in the logs.
partitionlevel	smallint		The level of this partition in the hierarchy.
parenttablename	name		The relation name of the parent table one level up from this partition.
parentschemaname	name		The name of the schema where the parent table resides.
parent_relid	oid		The OID of the parent table one level up from this partition.

pg_stat_replication

The pg_stat_replication view contains metadata of the walsender process that is used for Greenplum Database master mirroring.

Table 206: pg_catalog.pg_stat_replication

column	type	references	description
pid	integer		Process ID of WAL sender backend process.
usesysid	integer		User system ID that runs the WAL sender backend process
usename	name		User name that runs WAL sender backend process.
application_name	oid		Client application name.
client_addr	name		Client IP address.
client_port	integer		Client port number.
backend_start	timestamp		Operation start timestamp.
state	text		WAL sender state. The value can be: startup backup catchup streaming
sent_location	text		WAL sender xlog record sent location.
write_location	text		WAL receiver xlog record write location.
flush_location	text		WAL receiver xlog record flush location.
replay_location	text		Standby xlog record replay location.
sync_priority	text		Priority. the value is 1.
sync_state	text		WAL sender synchronization state. The value is sync.

pg_statistic

The pg_statistic system catalog table stores statistical data about the contents of the database. Entries are created by ANALYZE and subsequently used by the query optimizer. There is one entry for

each table column that has been analyzed. Note that all the statistical data is inherently approximate, even assuming that it is up-to-date.

`pg_statistic` also stores statistical data about the values of index expressions. These are described as if they were actual data columns; in particular, `starelid` references the index. No entry is made for an ordinary non-expression index column, however, since it would be redundant with the entry for the underlying table column. Currently, entries for index expressions always have `stainherit = false`.

When `stainherit = false`, there is normally one entry for each table column that has been analyzed. If the table has inheritance children, Greenplum Database creates a second entry with `stainherit = true`. This row represents the column's statistics over the inheritance tree, for example, statistics for the data you would see with `SELECT` *`column`* `FROM` *`table`*`*`, whereas the `stainherit = false` row represents the results of `SELECT` *`column`* `FROM ONLY` *`table`*.

Since different kinds of statistics may be appropriate for different kinds of data, `pg_statistic` is designed not to assume very much about what sort of statistics it stores. Only extremely general statistics (such as nullness) are given dedicated columns in `pg_statistic`. Everything else is stored in slots, which are groups of associated columns whose content is identified by a code number in one of the slot's columns.

Statistical information about a table's contents should be considered sensitive (for example: minimum and maximum values of a salary column). `pg_stats` is a publicly readable view on `pg_statistic` that only exposes information about those tables that are readable by the current user.

> **Warning:** Diagnostic tools such as `gpsd` and `minirepro` collect sensitive information from `pg_statistic`, such as histogram boundaries, in a clear, readable form. Always review the output files of these utilities to ensure that the contents are acceptable for transport outside of the database in your organization.

Table 207: pg_catalog.pg_statistic

column	type	references	description
`starelid`	oid	pg_class.oid	The table or index that the described column belongs to.
`staattnum`	int2	pg_attribute.attnum	The number of the described column.
`stainherit`	bool		If true, the statistics include inheritance child columns, not just the values in the specified relations.
`stanullfrac`	float4		The fraction of the column's entries that are null.
`stawidth`	int4		The average stored width, in bytes, of nonnull entries.

column	type	references	description
`stadistinct`	float4		The number of distinct nonnull data values in the column. A value greater than zero is the actual number of distinct values. A value less than zero is the negative of a fraction of the number of rows in the table (for example, a column in which values appear about twice on the average could be represented by `stadistinct` = -0.5). A zero value means the number of distinct values is unknown.
`stakind`*`N`*	int2		A code number indicating the kind of statistics stored in the `N`th slot of the `pg_statistic` row.
`staop`*`N`*	oid	pg_operator.oid	An operator used to derive the statistics stored in the `N`th slot. For example, a histogram slot would show the < operator that defines the sort order of the data.
`stanumbers`*`N`*	float4[]		Numerical statistics of the appropriate kind for the `N`th slot, or NULL if the slot kind does not involve numerical values.
`stavalues`*`N`*	anyarray		Column data values of the appropriate kind for the `N`th slot, or NULL if the slot kind does not store any data values. Each array's element values are actually of the specific column's data type, so there is no way to define these columns' type more specifically than `anyarray`.

pg_stat_resqueues

> **Note:** The `pg_stat_resqueues` view is valid only when resource queue-based resource management is active.

The `pg_stat_resqueues` view allows administrators to view metrics about a resource queue's workload over time. To allow statistics to be collected for this view, you must enable the `stats_queue_level` server configuration parameter on the Greenplum Database master instance. Enabling the collection of these metrics does incur a small performance penalty, as each statement submitted through a resource queue must be logged in the system catalog tables.

Table 208: pg_catalog.pg_stat_resqueues

column	type	references	description
`queueoid`	oid		The OID of the resource queue.
`queuename`	name		The name of the resource queue.
`n_queries_exec`	bigint		Number of queries submitted for execution from this resource queue.
`n_queries_wait`	bigint		Number of queries submitted to this resource queue that had to wait before they could execute.
`elapsed_exec`	bigint		Total elapsed execution time for statements submitted through this resource queue.
`elapsed_wait`	bigint		Total elapsed time that statements submitted through this resource queue had to wait before they were executed.

pg_tablespace

The `pg_tablespace` system catalog table stores information about the available tablespaces. Tables can be placed in particular tablespaces to aid administration of disk layout. Unlike most system catalogs, `pg_tablespace` is shared across all databases of a Greenplum system: there is only one copy of `pg_tablespace` per system, not one per database.

Table 209: pg_catalog.pg_tablespace

column	type	references	description
`spcname`	name		Tablespace name.
`spcowner`	oid	pg_authid.oid	Owner of the tablespace, usually the user who created it.

column	type	references	description
spcacl	aclitem[]		Tablespace access privileges.
spcoptions	text[]		Tablespace contentID locations.

pg_trigger

The pg_trigger system catalog table stores triggers on tables.

Note: Greenplum Database does not support triggers.

Table 210: pg_catalog.pg_trigger

column	type	references	description
tgrelid	oid	*pg_class.oid* Note that Greenplum Database does not enforce referential integrity.	The table this trigger is on.
tgname	name		Trigger name (must be unique among triggers of same table).
tgfoid	oid	*pg_proc.oid* Note that Greenplum Database does not enforce referential integrity.	The function to be called.
tgtype	int2		Bit mask identifying trigger conditions.
tgenabled	boolean		True if trigger is enabled.
tgisinternal	boolean		True if trigger is internally generated (usually, to enforce the constraint identified by tgconstraint).
tgconstrrelid	oid	*pg_class.oid* Note that Greenplum Database does not enforce referential integrity.	The table referenced by an referential integrity constraint.
tgdeferrable	boolean		True if deferrable.
tginitdeferred	boolean		True if initially deferred.
tgnargs	int2		Number of argument strings passed to trigger function.

column	type	references	description
tgattr	int2vector		Currently not used.
tgargs	bytea		Argument strings to pass to trigger, each NULL-terminated.

pg_type

The pg_type system catalog table stores information about data types. Base types (scalar types) are created with CREATE TYPE, and domains with CREATE DOMAIN. A composite type is automatically created for each table in the database, to represent the row structure of the table. It is also possible to create composite types with CREATE TYPE AS.

Table 211: pg_catalog.pg_type

column	type	references	description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
typname	name		Data type name
typnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this type
typowner	oid	pg_authid.oid	Owner of the type
typlen	int2		For a fixed-size type, typlen is the number of bytes in the internal representation of the type. But for a variable-length type, typlen is negative. -1 indicates a 'varlena' type (one that has a length word), -2 indicates a null-terminated C string.
typbyval	boolean		Determines whether internal routines pass a value of this type by value or by reference. typbyval had better be false if typlen is not 1, 2, or 4 (or 8 on machines where Datum is 8 bytes). Variable-length types are always passed by reference. Note that typbyval can be false even if the length would allow pass-by-value.

column	type	references	description
typtype	char		b for a base type, c for a composite type, d for a domain, e for an enum type, p for a pseudo-type, or r for a range type. See also typrelid and typbasetype.
typcategory	char		Arbitrary classification of data types that is used by the parser to determine which implicit casts should be preferred. See *Category Codes*.
typispreferred	boolean		True if the type is a preferred cast target within its typcategory
typisdefined	boolean		True if the type is defined, false if this is a placeholder entry for a not-yet-defined type. When false, nothing except the type name, namespace, and OID can be relied on.
typdelim	char		Character that separates two values of this type when parsing array input. Note that the delimiter is associated with the array element data type, not the array data type.
typrelid	oid	pg_class.oid	If this is a composite type (see typtype), then this column points to the pg_class entry that defines the corresponding table. (For a free-standing composite type, the pg_class entry does not really represent a table, but it is needed anyway for the type's pg_attribute entries to link to.) Zero for non-composite types.

column	type	references	description
typelem	oid	pg_type.oid	If not 0 then it identifies another row in pg_type. The current type can then be subscripted like an array yielding values of type typelem. A "true" array type is variable length (typlen = -1), but some fixed-length (typlen > 0) types also have nonzero typelem, for example name and point. If a fixed-length type has a typelem then its internal representation must be some number of values of the typelem data type with no other data. Variable-length array types have a header defined by the array subroutines.
typarray	oid	pg_type.oid	If not 0, identifies another row in pg_type, which is the "true" array type having this type as its element. Use pg_type.typarray to locate the array type associated with a specific type.
typinput	regproc	pg_proc.oid	Input conversion function (text format)
typoutput	regproc	pg_proc.oid	Output conversion function (text format)
typreceive	regproc	pg_proc.oid	Input conversion function (binary format), or 0 if none
typsend	regproc	pg_proc.oid	Output conversion function (binary format), or 0 if none
typmodin	regproc	pg_proc.oid	Type modifier input function, or 0 if the type does not support modifiers
typmodout	regproc	pg_proc.oid	Type modifier output function, or 0 to use the standard format

column	type	references	description
typanalyze	regproc	pg_proc.oid	Custom ANALYZE function, or 0 to use the standard function
typalign	char		The alignment required when storing a value of this type. It applies to storage on disk as well as most representations of the value inside Greenplum Database. When multiple values are stored consecutively, such as in the representation of a complete row on disk, padding is inserted before a datum of this type so that it begins on the specified boundary. The alignment reference is the beginning of the first datum in the sequence. Possible values are: c = char alignment (no alignment needed). s = short alignment (2 bytes on most machines). i = int alignment (4 bytes on most machines). d = double alignment (8 bytes on many machines, but not all).

column	type	references	description
typstorage	char		For varlena types (those with typlen = -1) tells if the type is prepared for toasting and what the default strategy for attributes of this type should be. Possible values are: p: Value must always be stored plain. e: Value can be stored in a secondary relation (if relation has one, see pg_class.reltoastrelid). m: Value can be stored compressed inline. x: Value can be stored compressed inline or stored in secondary storage. Note that m columns can also be moved out to secondary storage, but only as a last resort (e and x columns are moved first).
typnotnull	boolean		Represents a not-null constraint on a type. Used for domains only.
typbasetype	oid	pg_type.oid	Identifies the type that a domain is based on. Zero if this type is not a domain.
typtypmod	int4		Domains use typtypmod to record the typmod to be applied to their base type (-1 if base type does not use a typmod). -1 if this type is not a domain.

column	type	references	description
typndims	int4		The number of array dimensions for a domain over an array (if typbasetype is an array type). Zero for types other than domains over array types.
typcollation	oid	pg_collation.oid	Specifies the collation of the type. Zero if the type does not support collations. The value is DEFAULT_COLLATION_OID for a base type that supports collations. A domain over a collatable type can have some other collation OID if one was specified for the domain.
typdefaultbin	pg_node_tree		If not null, it is the nodeToString() representation of a default expression for the type. This is only used for domains.
typdefault	text		Null if the type has no associated default value. If typdefaultbin is not null, typdefault must contain a human-readable version of the default expression represented by typdefaultbin. If typdefaultbin is null and typdefault is not, then typdefault is the external representation of the type's default value, which may be fed to the type's input converter to produce a constant.
typacl	aclitem[]		Access privileges; see *GRANT* and *REVOKE* for details.

The following table lists the system-defined values of typcategory. Any future additions to this list will also be upper-case ASCII letters. All other ASCII characters are reserved for user-defined categories.

Table 212: typcategory Codes

Code	Category
A	Array types
B	Boolean types
C	Composite types
D	Date/time types
E	Enum types
G	Geometric types
I	Network address types
N	Numeric types
P	Pseudo-types
R	Range types
S	String types
T	Timespan types
U	User-defined types
V	Bit-string types
X	`unknown` type

pg_type_encoding

The `pg_type_encoding` system catalog table contains the column storage type information.

Table 213: pg_catalog.pg_type_encoding

column	type	modifers	storage	description
`typeid`	oid	not null	plain	Foreign key to *pg_attribute*
`typoptions`	text [ ]		extended	The actual options

pg_user_mapping

The system catalog table `pg_user_mapping` stores the mappings from local user to remote user. You must have administrator privileges to view this catalog. Access to this catalog is restricted from normal users, use the `pg_user_mappings` view instead.

Table 214: pg_catalog.pg_user_mapping

column	type	references	description
`umuser`	oid	pg_authid.oid	OID of the local role being mapped, 0 if the user mapping is public.

column	type	references	description
umserver	oid	pg_foreign_server.oid	OID of the foreign server that contains this mapping.
umoptions	text[]		User mapping-specific options, as "keyword= value" strings.

pg_user_mappings

The pg_user_mappings view provides access to information about user mappings. This view is essentially a public-readble view of the pg_user_mapping system catalog table that omits the options field if the user does not have access rights to view it.

Table 215: pg_user_mappings

column	type	references	description
umid	oid	pg_user_mapping.oid	OID of the user mapping.
srvid	oid	pg_foreign_server.oid	OID of the foreign server that contains this mapping.
srvname	text	pg_foreign_server. srvname	Name of the foreign server.
umuser	oid	pg_authid.oid	OID of the local role being mapped, 0 if the user mapping is public.
usename	name		Name of the local user to be mapped.
umoptions	text[]		User mapping-specific options, as "keyword= value" strings.

To protect password information stored as a user mapping option, the umoptions column reads as null unless one of the following applies:

- The current user is the user being mapped, and owns the server or holds USAGE privilege on it.
- The current user is the server owner and the mapping is for PUBLIC.
- The current user is a superuser.

user_mapping_options

The user_mapping_options view contains all of the options defined for user mappings in the current database. Greenplum Database displays only those user mappings to which the current user has access (by way of being the owner or having some privilege).

Table 216: user_mapping_options

column	type	references	description
`authorization_identifier`	sql_identifier		Name of the user being mapped, or `PUBLIC` if the mapping is public.
`foreign_server_catalog`	sql_identifier		Name of the database in which the foreign server used by this mapping is defined (always the current database).
`foreign_server_name`	sql_identifier		Name of the foreign server used by this mapping.
`option_name`	sql_identifier		Name of an option.
`option_value`	character_data		Value of the option. This column will display null unless: • The current user is the user being mapped. • The mapping is for `PUBLIC` and the current user is the foreign server owner. • The current user is a superuser. The intent is to protect password information stored as a user mapping option.

user_mappings

The `user_mappings` view contains all of the user mappgins defined in the current database. Greenplum Database displays only those user mappings to which the current user has access (by way of being the owner or having some privilege).

Table 217: user_mappings

column	type	references	description
`authorization_identifier`	sql_identifier		Name of the user being mapped, or `PUBLIC` if the mapping is public.
`foreign_server_catalog`	sql_identifier		Name of the database in which the foreign server used by this mapping is defined (always the current database).

column	type	references	description
`foreign_server_name`	sql_identifier		Name of the foreign server used by this mapping.

The gp_toolkit Administrative Schema

Greenplum Database provides an administrative schema called `gp_toolkit` that you can use to query the system catalogs, log files, and operating environment for system status information. The `gp_toolkit` schema contains a number of views that you can access using SQL commands. The `gp_toolkit` schema is accessible to all database users, although some objects may require superuser permissions. For convenience, you may want to add the `gp_toolkit` schema to your schema search path. For example:

```
=> ALTER ROLE myrole SET search_path TO myschema,gp_toolkit;
```

This documentation describes the most useful views in `gp_toolkit`. You may notice other objects (views, functions, and external tables) within the `gp_toolkit` schema that are not described in this documentation (these are supporting objects to the views described in this section).

> **Warning:** Do not change database objects in the gp_toolkit schema. Do not create database objects in the schema. Changes to objects in the schema might affect the accuracy of administrative information returned by schema objects. Any changes made in the gp_toolkit schema are lost when the database is backed up and then restored with the `gpbackup` and `gprestore` utilities.

These are the categories for views in the `gp_toolkit` schema.

Checking for Tables that Need Routine Maintenance

The following views can help identify tables that need routine table maintenance (`VACUUM` and/or `ANALYZE`).

- *gp_bloat_diag*
- *gp_stats_missing*

The `VACUUM` or `VACUUM FULL` command reclaims disk space occupied by deleted or obsolete rows. Because of the MVCC transaction concurrency model used in Greenplum Database, data rows that are deleted or updated still occupy physical space on disk even though they are not visible to any new transactions. Expired rows increase table size on disk and eventually slow down scans of the table.

The `ANALYZE` command collects column-level statistics needed by the query optimizer. Greenplum Database uses a cost-based query optimizer that relies on database statistics. Accurate statistics allow the query optimizer to better estimate selectivity and the number of rows retrieved by a query operation in order to choose the most efficient query plan.

gp_bloat_diag

This view shows regular heap-storage tables that have bloat (the actual number of pages on disk exceeds the expected number of pages given the table statistics). Tables that are bloated require a `VACUUM` or a `VACUUM FULL` in order to reclaim disk space occupied by deleted or obsolete rows. This view is accessible to all users, however non-superusers will only be able to see the tables that they have permission to access.

> **Note:** For diagnostic functions that return append-optimized table information, see *Checking Append-Optimized Tables*.

Table 218: gp_bloat_diag view

Column	Description
bdirelid	Table object id.

Column	Description
bdinspname	Schema name.
bdirelname	Table name.
bdirelpages	Actual number of pages on disk.
bdiexppages	Expected number of pages given the table data.
bdidiag	Bloat diagnostic message.

gp_stats_missing

This view shows tables that do not have statistics and therefore may require an `ANALYZE` be run on the table.

> **Note:** By default, `gp_stats_missing` does not display data for materialized views. Refer to *Including Data for Materialized Views* for instructions on adding this data to the `gp_stats_missing*` view output.

Table 219: gp_stats_missing view

Column	Description
smischema	Schema name.
smitable	Table name.
smisize	Does this table have statistics? False if the table does not have row count and row sizing statistics recorded in the system catalog, which may indicate that the table needs to be analyzed. This will also be false if the table does not contain any rows. For example, the parent tables of partitioned tables are always empty and will always return a false result.
smicols	Number of columns in the table.
smirecs	Number of rows in the table.

Checking for Locks

When a transaction accesses a relation (such as a table), it acquires a lock. Depending on the type of lock acquired, subsequent transactions may have to wait before they can access the same relation. For more information on the types of locks, see "Managing Data" in the *Greenplum Database Administrator Guide*. Greenplum Database resource queues (used for resource management) also use locks to control the admission of queries into the system.

The `gp_locks_*` family of views can help diagnose queries and sessions that are waiting to access an object due to a lock.

- *gp_locks_on_relation*
- *gp_locks_on_resqueue*

gp_locks_on_relation

This view shows any locks currently being held on a relation, and the associated session information about the query associated with the lock. For more information on the types of locks, see "Managing Data" in the *Greenplum Database Administrator Guide*. This view is accessible to all users, however non-superusers will only be able to see the locks for relations that they have permission to access.

Table 220: gp_locks_on_relation view

Column	Description
lorlocktype	Type of the lockable object: `relation`, `extend`, `page`, `tuple`, `transactionid`, `object`, `userlock`, `resource queue`, or `advisory`
lordatabase	Object ID of the database in which the object exists, zero if the object is a shared object.
lorrelname	The name of the relation.
lorrelation	The object ID of the relation.
lortransaction	The transaction ID that is affected by the lock.
lorpid	Process ID of the server process holding or awaiting this lock. NULL if the lock is held by a prepared transaction.
lormode	Name of the lock mode held or desired by this process.
lorgranted	Displays whether the lock is granted (true) or not granted (false).
lorcurrentquery	The current query in the session.

gp_locks_on_resqueue

> **Note:** The `gp_locks_on_resqueue` view is valid only when resource queue-based resource management is active.

This view shows any locks currently being held on a resource queue, and the associated session information about the query associated with the lock. This view is accessible to all users, however non-superusers will only be able to see the locks associated with their own sessions.

Table 221: gp_locks_on_resqueue view

Column	Description
lorusename	Name of the user executing the session.
lorrsqname	The resource queue name.
lorlocktype	Type of the lockable object: `resource queue`
lorobjid	The ID of the locked transaction.
lortransaction	The ID of the transaction that is affected by the lock.
lorpid	The process ID of the transaction that is affected by the lock.
lormode	The name of the lock mode held or desired by this process.
lorgranted	Displays whether the lock is granted (true) or not granted (false).
lorwaiting	Displays whether or not the session is waiting.

Checking Append-Optimized Tables

The `gp_toolkit` schema includes a set of diagnostic functions you can use to investigate the state of append-optimized tables.

When an append-optimized table (or column-oriented append-optimized table) is created, another table is implicitly created, containing metadata about the current state of the table. The metadata includes information such as the number of records in each of the table's segments.

Append-optimized tables may have non-visible rows—rows that have been updated or deleted, but remain in storage until the table is compacted using `VACUUM`. The hidden rows are tracked using an auxiliary visibility map table, or visimap.

The following functions let you access the metadata for append-optimized and column-oriented tables and view non-visible rows.

For most of the functions, the input argument is `regclass`, either the table `name` or the `oid` of a table.

__gp_aovisimap_compaction_info(oid)

This function displays compaction information for an append-optimized table. The information is for the on-disk data files on Greenplum Database segments that store the table data. You can use the information to determine the data files that will be compacted by a `VACUUM` operation on an append-optimized table.

> **Note:** Until a `VACUUM` operation deletes the row from the data file, deleted or updated data rows occupy physical space on disk even though they are hidden to new transactions. The configuration parameter *`gp_appendonly_compaction`* controls the functionality of the `VACUUM` command.

This table describes the `__gp_aovisimap_compaction_info` function output table.

Table 222: __gp_aovisimap_compaction_info output table

Column	Description
content	Greenplum Database segment ID.
datafile	ID of the data file on the segment.
compaction_possible	The value is either `t` or `f`. The value `t` indicates that the data in data file be compacted when a `VACUUM` operation is performed. The server configuration parameter `gp_appendonly_compaction_threshold` affects this value.
hidden_tupcount	In the data file, the number of hidden (deleted or updated) rows.
total_tupcount	In the data file, the total number of rows.
percent_hidden	In the data file, the ratio (as a percentage) of hidden (deleted or updated) rows to total rows.

__gp_aoseg(regclass)

This function returns metadata information contained in the append-optimized table's on-disk segment file.

The input argument is the name or the oid of an append-optimized table.

Table 223: __gp_aoseg_name output table

Column	Description
segno	The file segment number.
eof	The effective end of file for this file segment.
tupcount	The total number of tuples in the segment, including invisible tuples.
varblockcount	The total number of varblocks in the file segment.
eof_uncompressed	The end of file if the file segment were uncompressed.
modcount	The number of data modification operations.
state	The state of the file segment. Indicates if the segment is active or ready to be dropped after compaction.

__gp_aoseg_history(regclass)

This function returns metadata information contained in the append-optimized table's on-disk segment file. It displays all different versions (heap tuples) of the aoseg meta information. The data is complex, but users with a deep understanding of the system may find it useful for debugging.

The input argument is the name or the oid of an append-optimized table.

Table 224: __gp_aoseg_history output table

Column	Description
gp_tid	The id of the tuple.
gp_xmin	The id of the earliest transaction.
gp_xmin_status	Status of the gp_xmin transaction.
gp_xmin_commit_	The commit distribution id of the gp_xmin transaction.
gp_xmax	The id of the latest transaction.
gp_xmax_status	The status of the latest transaction.
gp_xmax_commit_	The commit distribution id of the gp_xmax transaction.
gp_command_id	The id of the query command.
gp_infomask	A bitmap containing state information.
gp_update_tid	The ID of the newer tuple if the row is updated.
gp_visibility	The tuple visibility status.
segno	The number of the segment in the segment file.
tupcount	The number of tuples, including hidden tuples.
eof	The effective end of file for the segment.

Column	Description
eof_uncompressed	The end of file for the segment if data were uncompressed.
modcount	A count of data modifications.
state	The status of the segment.

__gp_aocsseg(regclass)

This function returns metadata information contained in a column-oriented append-optimized table's on-disk segment file, excluding non-visible rows. Each row describes a segment for a column in the table.

The input argument is the name or the oid of a column-oriented append-optimized table.

Table 225: __gp_aocsseg(oid) output table

Column	Description
gp_tid	The table id.
segno	The segment number.
column_num	The column number.
physical_segno	The number of the segment in the segment file.
tupcount	The number of rows in the segment, excluding hidden tuples.
eof	The effective end of file for the segment.
eof_uncompressed	The end of file for the segment if the data were uncompressed.
modcount	A count of data modification operations for the segment.
state	The status of the segment.

__gp_aocsseg_history(regclass)

This function returns metadata information contained in a column-oriented append-optimized table's on-disk segment file. Each row describes a segment for a column in the table. The data is complex, but users with a deep understanding of the system may find it useful for debugging.

The input argument is the name or the oid of a column-oriented append-optimized table.

Table 226: __gp_aocsseg_history output table

Column	Description
gp_tid	The oid of the tuple.
gp_xmin	The earliest transaction.
gp_xmin_status	The status of the gp_xmin transaction.
gp_xmin_	Text representation of gp_xmin.
gp_xmax	The latest transaction.
gp_xmax_status	The status of the gp_xmax transaction.

Column	Description
gp_xmax_	Text representation of gp_max.
gp_command_id	ID of the command operating on the tuple.
gp_infomask	A bitmap containing state information.
gp_update_tid	The ID of the newer tuple if the row is updated.
gp_visibility	The tuple visibility status.
segno	The segment number in the segment file.
column_num	The column number.
physical_segno	The segment containing data for the column.
tupcount	The total number of tuples in the segment.
eof	The effective end of file for the segment.
eof_uncompressed	The end of file for the segment if the data were uncompressed.
modcount	A count of the data modification operations.
state	The state of the segment.

__gp_aovisimap(regclass)

This function returns the tuple ID, the segment file, and the row number of each non-visible tuple according to the visibility map.

The input argument is the name or the oid of an append-optimized table.

Column	Description
tid	The tuple id.
segno	The number of the segment file.
row_num	The row number of a row that has been deleted or updated.

__gp_aovisimap_hidden_info(regclass)

This function returns the numbers of hidden and visible tuples in the segment files for an append-optimized table.

The input argument is the name or the oid of an append-optimized table.

Column	Description
segno	The number of the segment file.
hidden_tupcount	The number of hidden tuples in the segment file.
total_tupcount	The total number of tuples in the segment file.

__gp_aovisimap_entry(regclass)

This function returns information about each visibility map entry for the table.

The input argument is the name or the oid of an append-optimized table.

Table 227: __gp_aovisimap_entry output table

Column	Description
segno	Segment number of the visibility map entry.
first_row_num	The first row number of the entry.
hidden_tupcount	The number of hidden tuples in the entry.
bitmap	A text representation of the visibility bitmap.

Viewing Greenplum Database Server Log Files

Each component of a Greenplum Database system (master, standby master, primary segments, and mirror segments) keeps its own server log files. The `gp_log_*` family of views allows you to issue SQL queries against the server log files to find particular entries of interest. The use of these views require superuser permissions.

- *gp_log_command_timings*
- *gp_log_database*
- *gp_log_master_concise*
- *gp_log_system*

gp_log_command_timings

This view uses an external table to read the log files on the master and report the execution time of SQL commands executed in a database session. The use of this view requires superuser permissions.

Table 228: gp_log_command_timings view

Column	Description
logsession	The session identifier (prefixed with "con").
logcmdcount	The command number within a session (prefixed with "cmd").
logdatabase	The name of the database.
loguser	The name of the database user.
logpid	The process id (prefixed with "p").
logtimemin	The time of the first log message for this command.
logtimemax	The time of the last log message for this command.
logduration	Statement duration from start to end time.

gp_log_database

This view uses an external table to read the server log files of the entire Greenplum system (master, segments, and mirrors) and lists log entries associated with the current database. Associated log entries can be identified by the session id (logsession) and command id (logcmdcount). The use of this view requires superuser permissions.

Table 229: gp_log_database view

Column	Description
logtime	The timestamp of the log message.
loguser	The name of the database user.
logdatabase	The name of the database.
logpid	The associated process id (prefixed with "p").
logthread	The associated thread count (prefixed with "th").
loghost	The segment or master host name.
logport	The segment or master port.
logsessiontime	Time session connection was opened.
logtransaction	Global transaction id.
logsession	The session identifier (prefixed with "con").
logcmdcount	The command number within a session (prefixed with "cmd").
logsegment	The segment content identifier (prefixed with "seg" for primary or "mir" for mirror. The master always has a content id of -1).
logslice	The slice id (portion of the query plan being executed).
logdistxact	Distributed transaction id.
loglocalxact	Local transaction id.
logsubxact	Subtransaction id.
logseverity	LOG, ERROR, FATAL, PANIC, DEBUG1 or DEBUG2.
logstate	SQL state code associated with the log message.
logmessage	Log or error message text.
logdetail	Detail message text associated with an error message.
loghint	Hint message text associated with an error message.
logquery	The internally-generated query text.
logquerypos	The cursor index into the internally-generated query text.
logcontext	The context in which this message gets generated.
logdebug	Query string with full detail for debugging.
logcursorpos	The cursor index into the query string.
logfunction	The function in which this message is generated.
logfile	The log file in which this message is generated.

Column	Description
logline	The line in the log file in which this message is generated.
logstack	Full text of the stack trace associated with this message.

gp_log_master_concise

This view uses an external table to read a subset of the log fields from the master log file. The use of this view requires superuser permissions.

Table 230: gp_log_master_concise view

Column	Description
logtime	The timestamp of the log message.
logdatabase	The name of the database.
logsession	The session identifier (prefixed with "con").
logcmdcount	The command number within a session (prefixed with "cmd").
logmessage	Log or error message text.

gp_log_system

This view uses an external table to read the server log files of the entire Greenplum system (master, segments, and mirrors) and lists all log entries. Associated log entries can be identified by the session id (logsession) and command id (logcmdcount). The use of this view requires superuser permissions.

Table 231: gp_log_system view

Column	Description
logtime	The timestamp of the log message.
loguser	The name of the database user.
logdatabase	The name of the database.
logpid	The associated process id (prefixed with "p").
logthread	The associated thread count (prefixed with "th").
loghost	The segment or master host name.
logport	The segment or master port.
logsessiontime	Time session connection was opened.
logtransaction	Global transaction id.
logsession	The session identifier (prefixed with "con").
logcmdcount	The command number within a session (prefixed with "cmd").

Column	Description
logsegment	The segment content identifier (prefixed with "seg" for primary or "mir" for mirror. The master always has a content id of -1).
logslice	The slice id (portion of the query plan being executed).
logdistxact	Distributed transaction id.
loglocalxact	Local transaction id.
logsubxact	Subtransaction id.
logseverity	LOG, ERROR, FATAL, PANIC, DEBUG1 or DEBUG2.
logstate	SQL state code associated with the log message.
logmessage	Log or error message text.
logdetail	Detail message text associated with an error message.
loghint	Hint message text associated with an error message.
logquery	The internally-generated query text.
logquerypos	The cursor index into the internally-generated query text.
logcontext	The context in which this message gets generated.
logdebug	Query string with full detail for debugging.
logcursorpos	The cursor index into the query string.
logfunction	The function in which this message is generated.
logfile	The log file in which this message is generated.
logline	The line in the log file in which this message is generated.
logstack	Full text of the stack trace associated with this message.

Checking Server Configuration Files

Each component of a Greenplum Database system (master, standby master, primary segments, and mirror segments) has its own server configuration file (`postgresql.conf`). The following `gp_toolkit` objects can be used to check parameter settings across all primary `postgresql.conf` files in the system:

- *gp_param_setting('parameter_name')*
- *gp_param_settings_seg_value_diffs*

gp_param_setting('parameter_name')

This function takes the name of a server configuration parameter and returns the `postgresql.conf` value for the master and each active segment. This function is accessible to all users.

Table 232: gp_param_setting('parameter_name') function

Column	Description
paramsegment	The segment content id (only active segments are shown). The master content id is always -1.
paramname	The name of the parameter.
paramvalue	The value of the parameter.

Example:

```
SELECT * FROM gp_param_setting('max_connections');
```

gp_param_settings_seg_value_diffs

Server configuration parameters that are classified as *local* parameters (meaning each segment gets the parameter value from its own `postgresql.conf` file), should be set identically on all segments. This view shows local parameter settings that are inconsistent. Parameters that are supposed to have different values (such as `port`) are not included. This view is accessible to all users.

Table 233: gp_param_settings_seg_value_diffs view

Column	Description
psdname	The name of the parameter.
psdvalue	The value of the parameter.
psdcount	The number of segments that have this value.

Checking for Failed Segments

The *gp_pgdatabase_invalid* view can be used to check for down segments.

gp_pgdatabase_invalid

This view shows information about segments that are marked as down in the system catalog. This view is accessible to all users.

Table 234: gp_pgdatabase_invalid view

Column	Description
pgdbidbid	The segment dbid. Every segment has a unique dbid.
pgdbiisprimary	Is the segment currently acting as the primary (active) segment? (t or f)
pgdbicontent	The content id of this segment. A primary and mirror will have the same content id.
pgdbivalid	Is this segment up and valid? (t or f)
pgdbidefinedprimary	Was this segment assigned the role of primary at system initialization time? (t or f)

Checking Resource Group Activity and Status

Note: The resource group activity and status views described in this section are valid only when resource group-based resource management is active.

Resource groups manage transactions to avoid exhausting system CPU and memory resources. Every database user is assigned a resource group. Greenplum Database evaluates every transaction submitted by a user against the limits configured for the user's resource group before running the transaction.

You can use the `gp_resgroup_config` view to check the configuration of each resource group. You can use the `gp_resgroup_status*` views to display the current transaction status and resource usage of each resource group.

- *gp_resgroup_config*
- *gp_resgroup_status*
- *gp_resgroup_status_per_host*
- *gp_resgroup_status_per_segment*

gp_resgroup_config

The `gp_resgroup_config` view allows administrators to see the current CPU, memory, and concurrency limits for a resource group.

This view is accessible to all users.

Table 235: gp_resgroup_config

Column	Description
groupid	The ID of the resource group.
groupname	The name of the resource group.
concurrency	The concurrency (`CONCURRENCY`) value specified for the resource group.
cpu_rate_limit	The CPU limit (`CPU_RATE_LIMIT`) value specified for the resource group, or -1.
memory_limit	The memory limit (`MEMORY_LIMIT`) value specified for the resource group.
memory_shared_quota	The shared memory quota (`MEMORY_SHARED_QUOTA`) value specified for the resource group.
memory_spill_ratio	The memory spill ratio (`MEMORY_SPILL_RATIO`) value specified for the resource group.
memory_auditor	The memory auditor for the resource group.
cpuset	The CPU cores reserved for the resource group, or -1.

gp_resgroup_status

The `gp_resgroup_status` view allows administrators to see status and activity for a resource group. It shows how many queries are waiting to run and how many queries are currently active in the system for each resource group. The view also displays current memory and CPU usage for the resource group.

Note: Resource groups use the Linux control groups (cgroups) configured on the host systems. The cgroups are used to manage host system resources. When resource groups use cgroups

that are as part of a nested set of cgroups, resource group limits are relative to the parent cgroup allotment. For information about nested cgroups and Greenplum Database resource group limits, see *Understanding Role and Component Resource Groups*.

This view is accessible to all users.

Table 236: gp_resgroup_status view

Column	Description
rsgname	The name of the resource group.
groupid	The ID of the resource group.
num_running	The number of transactions currently executing in the resource group.
num_queueing	The number of currently queued transactions for the resource group.
num_queued	The total number of queued transactions for the resource group since the Greenplum Database cluster was last started, excluding the num_queueing.
num_executed	The total number of executed transactions in the resource group since the Greenplum Database cluster was last started, excluding the num_running.
total_queue_duration	The total time any transaction was queued since the Greenplum Database cluster was last started.
cpu_usage	A set of key-value pairs. For each segment instance (the key), the value is the real-time, per-segment instance CPU core usage by a resource group. The value is the sum of the percentages (as a decimal value) of CPU cores that are used by the resource group for the segment instance.
memory_usage	The real-time memory usage of the resource group on each Greenplum Database segment's host.

The `cpu_usage` field is a JSON-formatted, key:value string that identifies, for each resource group, the per-segment instance CPU core usage. The key is the segment id. The value is the sum of the percentages (as a decimal value) of the CPU cores used by the segment instance's resource group on the segment host; the maximum value is 1.00. The total CPU usage of all segment instances running on a host should not exceed the `gp_resource_group_cpu_limit`. Example `cpu_usage` column output:

```
{"-1":0.01, "0":0.31, "1":0.31}
```

In the example, segment `0` and segment `1` are running on the same host; their CPU usage is the same.

The `memory_usage` field is also a JSON-formatted, key:value string. The string contents differ depending upon the type of resource group. For each resource group that you assign to a role (default memory auditor `vmtracker`), this string identifies the used and available fixed and shared memory quota allocations on each segment. The key is segment id. The values are memory values displayed in MB units.

The following example shows `memory_usage` column output for a single segment for a resource group that you assign to a role:

```
"0":{"used":0, "available":76, "quota_used":-1, "quota_available":60,
 "shared_used":0, "shared_available":16}
```

For each resource group that you assign to an external component, the `memory_usage` JSON-formatted string identifies the memory used and the memory limit on each segment. The following example shows `memory_usage` column output for an external component resource group for a single segment:

```
"1":{"used":11, "limit_granted":15}
```

> **Note:** See the `gp_resgroup_status_per_host` and `gp_resgroup_status_per_segment` views, described below, for more user-friendly display of CPU and memory usage.

gp_resgroup_status_per_host

The *`gp_resgroup_status_per_host`* view displays the real-time CPU and memory usage (MBs) for each resource group on a per-host basis. The view also displays available and granted group fixed and shared memory for each resource group on a host.

Table 237: gp_resgroup_status_per_host view

Column	Description
`rsgname`	The name of the resource group.
`groupid`	The ID of the resource group.
`hostname`	The hostname of the segment host.
`cpu`	The real-time CPU core usage by the resource group on a host. The value is the sum of the percentages (as a decimal value) of the CPU cores that are used by the resource group on the host.
`memory_used`	The real-time memory usage of the resource group on the host. This total includes resource group fixed and shared memory. It also includes global shared memory used by the resource group.
`memory_available`	The unused fixed and shared memory for the resource group that is available on the host. This total does not include available resource group global shared memory.
`memory_quota_used`	The real-time fixed memory usage for the resource group on the host.
`memory_quota_available`	The fixed memory available to the resource group on the host.
`memory_shared_used`	The group shared memory used by the resource group on the host. If any global shared memory is used by the resource group, this amount is included in the total as well.
`memory_shared_available`	The amount of group shared memory available to the resource group on the host. Resource group global shared memory is not included in this total.

Sample output for the `gp_resgroup_status_per_host` view:

```
 rsgname       | groupid | hostname   | cpu  | memory_used |
 memory_available | memory_quota_used | memory_quota_available |
 memory_shared_used | memory_shared_available
---------------+---------+------------+------+-------------
+-----------------+------------------+-----------------------
+---------------------+---------------------
 admin_group   | 6438    | my-desktop | 0.84 | 1           | 271
  | 68                | 68                     | 0                   | 136

 default_group | 6437    | my-desktop | 0.00 | 0           | 816
  | 0                 | 400                    | 0                   | 416

(2 rows)
```

gp_resgroup_status_per_segment

The *gp_resgroup_status_per_segment* view displays the real-time CPU and memory usage (MBs) for each resource group on a per-segment-instance and per-host basis. The view also displays available and granted group fixed and shared memory for each resource group and segment instance combination on the host.

Table 238: gp_resgroup_status_per_segment view

Column	Description
rsgname	The name of the resource group.
groupid	The ID of the resource group.
hostname	The hostname of the segment host.
segment_id	The content ID for a segment instance on the segment host.
cpu	The real-time, per-segment instance CPU core usage by the resource group on the host. The value is the sum of the percentages (as a decimal value) of the CPU cores that are used by the resource group for the segment instance.
memory_used	The real-time memory usage of the resource group for the segment instance on the host. This total includes resource group fixed and shared memory. It also includes global shared memory used by the resource group.
memory_available	The unused fixed and shared memory for the resource group for the segment instance on the host.
memory_quota_used	The real-time fixed memory usage for the resource group for the segment instance on the host.
memory_quota_available	The fixed memory available to the resource group for the segment instance on the host.
memory_shared_used	The group shared memory used by the resource group for the segment instance on the host.

Column	Description
memory_shared_available	The amount of group shared memory available for the segment instance on the host. Resource group global shared memory is not included in this total.

Query output for this view is similar to that of the gp_resgroup_status_per_host view, and breaks out the CPU and memory (used and available) for each segment instance on each host.

Checking Resource Queue Activity and Status

Note: The resource queue activity and status views described in this section are valid only when resource queue-based resource management is active.

The purpose of resource queues is to limit the number of active queries in the system at any given time in order to avoid exhausting system resources such as memory, CPU, and disk I/O. All database users are assigned to a resource queue, and every statement submitted by a user is first evaluated against the resource queue limits before it can run. The gp_resq_* family of views can be used to check the status of statements currently submitted to the system through their respective resource queue. Note that statements issued by superusers are exempt from resource queuing.

- *gp_resq_activity*
- *gp_resq_activity_by_queue*
- *gp_resq_priority_statement*
- *gp_resq_role*
- *gp_resqueue_status*

gp_resq_activity

For the resource queues that have active workload, this view shows one row for each active statement submitted through a resource queue. This view is accessible to all users.

Table 239: gp_resq_activity view

Column	Description
resqprocpid	Process ID assigned to this statement (on the master).
resqrole	User name.
resqoid	Resource queue object id.
resqname	Resource queue name.
resqstart	Time statement was issued to the system.
resqstatus	Status of statement: running, waiting or cancelled.

gp_resq_activity_by_queue

For the resource queues that have active workload, this view shows a summary of queue activity. This view is accessible to all users.

Table 240: gp_resq_activity_by_queue Column

Column	Description
resqoid	Resource queue object id.

Column	Description
resqname	Resource queue name.
resqlast	Time of the last statement issued to the queue.
resqstatus	Status of last statement: running, waiting or cancelled.
resqtotal	Total statements in this queue.

gp_resq_priority_statement

This view shows the resource queue priority, session ID, and other information for all statements currently running in the Greenplum Database system. This view is accessible to all users.

Table 241: gp_resq_priority_statement view

Column	Description
rqpdatname	The database name that the session is connected to.
rqpusename	The user who issued the statement.
rqpsession	The session ID.
rqpcommand	The number of the statement within this session (the command id and session id uniquely identify a statement).
rqppriority	The resource queue priority for this statement (MAX, HIGH, MEDIUM, LOW).
rqpweight	An integer value associated with the priority of this statement.
rqpquery	The query text of the statement.

gp_resq_role

This view shows the resource queues associated with a role. This view is accessible to all users.

Table 242: gp_resq_role view

Column	Description
rrrolname	Role (user) name.
rrrsqname	The resource queue name assigned to this role. If a role has not been explicitly assigned to a resource queue, it will be in the default resource queue (*pg_default*).

gp_resqueue_status

This view allows administrators to see status and activity for a resource queue. It shows how many queries are waiting to run and how many queries are currently active in the system from a particular resource queue.

Table 243: gp_resqueue_status view

Column	Description
queueid	The ID of the resource queue.
rsqname	The name of the resource queue.
rsqcountlimit	The active query threshold of the resource queue. A value of -1 means no limit.
rsqcountvalue	The number of active query slots currently being used in the resource queue.
rsqcostlimit	The query cost threshold of the resource queue. A value of -1 means no limit.
rsqcostvalue	The total cost of all statements currently in the resource queue.
rsqmemorylimit	The memory limit for the resource queue.
rsqmemoryvalue	The total memory used by all statements currently in the resource queue.
rsqwaiters	The number of statements currently waiting in the resource queue.
rsqholders	The number of statements currently running on the system from this resource queue.

Checking Query Disk Spill Space Usage

The *gp_workfile_** views show information about all the queries that are currently using disk spill space. Greenplum Database creates work files on disk if it does not have sufficient memory to execute the query in memory. This information can be used for troubleshooting and tuning queries. The information in the views can also be used to specify the values for the Greenplum Database configuration parameters `gp_workfile_limit_per_query` and `gp_workfile_limit_per_segment`.

- *gp_workfile_entries*
- *gp_workfile_usage_per_query*
- *gp_workfile_usage_per_segment*

gp_workfile_entries

This view contains one row for each operator using disk space for workfiles on a segment at the current time. The view is accessible to all users, however non-superusers only to see information for the databases that they have permission to access.

Table 244: gp_workfile_entries

Column	Type	References	Description
datname	name		Greenplum database name.
pid	integer		Process ID of the server process.
sess_id	integer		Session ID.

Column	Type	References	Description
command_cnt	integer		Command ID of the query.
usename	name		Role name.
query	text		Current query that the process is running.
segid	integer		Segment ID.
slice	integer		The query plan slice. The portion of the query plan that is being executed.
optype	text		The query operator type that created the work file.
size	bigint		The size of the work file in bytes.
numfiles	integer		The number of files created.
prefix	text		Prefix used when naming a related set of workfiles.

gp_workfile_usage_per_query

This view contains one row for each query using disk space for workfiles on a segment at the current time. The view is accessible to all users, however non-superusers only to see information for the databases that they have permission to access.

Table 245: gp_workfile_usage_per_query

Column	Type	References	Description
datname	name		Greenplum database name.
pid	integer		Process ID of the server process.
sess_id	integer		Session ID.
command_cnt	integer		Command ID of the query.
usename	name		Role name.
query	text		Current query that the process is running.
segid	integer		Segment ID.
size	numeric		The size of the work file in bytes.

Column	Type	References	Description
numfiles	bigint		The number of files created.

gp_workfile_usage_per_segment

This view contains one row for each segment. Each row displays the total amount of disk space used for workfiles on the segment at the current time. The view is accessible to all users, however non-superusers only to see information for the databases that they have permission to access.

Table 246: gp_workfile_usage_per_segment

Column	Type	References	Description
segid	smallint		Segment ID.
size	numeric		The total size of the work files on a segment.
numfiles	bigint		The number of files created.

Viewing Users and Groups (Roles)

It is frequently convenient to group users (roles) together to ease management of object privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In Greenplum Database this is done by creating a role that represents the group, and then granting membership in the group role to individual user roles.

The *gp_roles_assigned* view can be used to see all of the roles in the system, and their assigned members (if the role is also a group role).

gp_roles_assigned

This view shows all of the roles in the system, and their assigned members (if the role is also a group role). This view is accessible to all users.

Table 247: gp_roles_assigned view

Column	Description
raroleid	The role object ID. If this role has members (users), it is considered a *group* role.
rarolename	The role (user or group) name.
rarememberid	The role object ID of the role that is a member of this role.
rarembername	Name of the role that is a member of this role.

Checking Database Object Sizes and Disk Space

The gp_size_* family of views can be used to determine the disk space usage for a distributed Greenplum Database, schema, table, or index. The following views calculate the total size of an object across all primary segments (mirrors are not included in the size calculations).

Note: By default, the `gp_size_*` views do not display data for materialized views. Refer to *Including Data for Materialized Views* for instructions on adding this data to `gp_size_*` view output.

- *gp_size_of_all_table_indexes*
- *gp_size_of_database*
- *gp_size_of_index*
- *gp_size_of_partition_and_indexes_disk*
- *gp_size_of_schema_disk*
- *gp_size_of_table_and_indexes_disk*
- *gp_size_of_table_and_indexes_licensing*
- *gp_size_of_table_disk*
- *gp_size_of_table_uncompressed*
- *gp_disk_free*

The table and index sizing views list the relation by object ID (not by name). To check the size of a table or index by name, you must look up the relation name (`relname`) in the `pg_class` table. For example:

```
SELECT relname as name, sotdsize as size, sotdtoastsize as
toast, sotdadditionalsize as other
FROM gp_size_of_table_disk as sotd, pg_class
WHERE sotd.sotdoid=pg_class.oid ORDER BY relname;
```

gp_size_of_all_table_indexes

This view shows the total size of all indexes for a table. This view is accessible to all users, however non-superusers will only be able to see relations that they have permission to access.

Table 248: gp_size_of_all_table_indexes view

Column	Description
soatioid	The object ID of the table
soatisize	The total size of all table indexes in bytes
soatischemaname	The schema name
soatitablename	The table name

gp_size_of_database

This view shows the total size of a database. This view is accessible to all users, however non-superusers will only be able to see databases that they have permission to access.

Table 249: gp_size_of_database view

Column	Description
sodddatname	The name of the database
sodddatsize	The size of the database in bytes

gp_size_of_index

This view shows the total size of an index. This view is accessible to all users, however non-superusers will only be able to see relations that they have permission to access.

Table 250: gp_size_of_index view

Column	Description
soioid	The object ID of the index
soitableoid	The object ID of the table to which the index belongs
soisize	The size of the index in bytes
soiindexschemaname	The name of the index schema
soiindexname	The name of the index
soitableschemaname	The name of the table schema
soitablename	The name of the table

gp_size_of_partition_and_indexes_disk

This view shows the size on disk of partitioned child tables and their indexes. This view is accessible to all users, however non-superusers will only be able to see relations that they have permission to access.

Table 251: gp_size_of_partition_and_indexes_disk view

Column	Description
sopaidparentoid	The object ID of the parent table
sopaidpartitionoid	The object ID of the partition table
sopaidpartitiontablesize	The partition table size in bytes
sopaidpartitionindexessize	The total size of all indexes on this partition
Sopaidparentschemaname	The name of the parent schema
Sopaidparenttablename	The name of the parent table
Sopaidpartitionschemaname	The name of the partition schema
sopaidpartitiontablename	The name of the partition table

gp_size_of_schema_disk

This view shows schema sizes for the public schema and the user-created schemas in the current database. This view is accessible to all users, however non-superusers will be able to see only the schemas that they have permission to access.

Table 252: gp_size_of_schema_disk view

Column	Description
sosdnsp	The name of the schema
sosdschematablesize	The total size of tables in the schema in bytes
sosdschemaidxsize	The total size of indexes in the schema in bytes

gp_size_of_table_and_indexes_disk

This view shows the size on disk of tables and their indexes. This view is accessible to all users, however non-superusers will only be able to see relations that they have permission to access.

Table 253: gp_size_of_table_and_indexes_disk view

Column	Description
sotaidoid	The object ID of the parent table
sotaidtablesize	The disk size of the table
sotaididxsize	The total size of all indexes on the table
sotaidschemaname	The name of the schema
sotaidtablename	The name of the table

gp_size_of_table_and_indexes_licensing

This view shows the total size of tables and their indexes for licensing purposes. The use of this view requires superuser permissions.

Table 254: gp_size_of_table_and_indexes_licensing view

Column	Description
sotailoid	The object ID of the table
sotailtablesizedisk	The total disk size of the table
sotailtablesizeuncompressed	If the table is a compressed append-optimized table, shows the uncompressed table size in bytes.
sotailindexessize	The total size of all indexes in the table
sotailschemaname	The schema name
sotailtablename	The table name

gp_size_of_table_disk

This view shows the size of a table on disk. This view is accessible to all users, however non-superusers will only be able to see tables that they have permission to access

Table 255: gp_size_of_table_disk view

Column	Description
sotdoid	The object ID of the table
sotdsize	The size of the table in bytes. The size is only the main table size. The size does not include auxiliary objects such as oversized (toast) attributes, or additional storage objects for AO tables.
sotdtoastsize	The size of the TOAST table (oversized attribute storage), if there is one.

Column	Description
sotdadditionalsize	Reflects the segment and block directory table sizes for append-optimized (AO) tables.
sotdschemaname	The schema name
sotdtablename	The table name

gp_size_of_table_uncompressed

This view shows the uncompressed table size for append-optimized (AO) tables. Otherwise, the table size on disk is shown. The use of this view requires superuser permissions.

Table 256: gp_size_of_table_uncompressed view

Column	Description
sotuoid	The object ID of the table
sotusize	The uncomressed size of the table in bytes if it is a compressed AO table. Otherwise, the table size on disk.
sotuschemaname	The schema name
sotutablename	The table name

gp_disk_free

This external table runs the `df` (disk free) command on the active segment hosts and reports back the results. Inactive mirrors are not included in the calculation. The use of this external table requires superuser permissions.

Table 257: gp_disk_free external table

Column	Description
dfsegment	The content id of the segment (only active segments are shown)
dfhostname	The hostname of the segment host
dfdevice	The device name
dfspace	Free disk space in the segment file system in kilobytes

Checking for Uneven Data Distribution

All tables in Greenplum Database are distributed, meaning their data is divided across all of the segments in the system. If the data is not distributed evenly, then query processing performance may suffer. The following views can help diagnose if a table has uneven data distribution:

- *gp_skew_coefficients*
- *gp_skew_idle_fractions*

> **Note:** By default, the `gp_skew_*` views do not display data for materialized views. Refer to *Including Data for Materialized Views* for instructions on adding this data to `gp_skew_*` view output.

gp_skew_coefficients

This view shows data distribution skew by calculating the coefficient of variation (CV) for the data stored on each segment. This view is accessible to all users, however non-superusers will only be able to see tables that they have permission to access

Table 258: gp_skew_coefficients view

Column	Description
skcoid	The object id of the table.
skcnamespace	The namespace where the table is defined.
skcrelname	The table name.
skccoeff	The coefficient of variation (CV) is calculated as the standard deviation divided by the average. It takes into account both the average and variability around the average of a data series. The lower the value, the better. Higher values indicate greater data skew.

gp_skew_idle_fractions

This view shows data distribution skew by calculating the percentage of the system that is idle during a table scan, which is an indicator of processing data skew. This view is accessible to all users, however non-superusers will only be able to see tables that they have permission to access

Table 259: gp_skew_idle_fractions view

Column	Description
sifoid	The object id of the table.
sifnamespace	The namespace where the table is defined.
sifrelname	The table name.
siffraction	The percentage of the system that is idle during a table scan, which is an indicator of uneven data distribution or query processing skew. For example, a value of 0.1 indicates 10% skew, a value of 0.5 indicates 50% skew, and so on. Tables that have more than 10% skew should have their distribution policies evaluated.

Including Data for Materialized Views

You must update a `gp_toolkit` internal view if you want data about materialized views to be included in the output of relevant `gp_toolkit` views.

Run the following SQL commands as the Greenplum Database administrator to update the internal view:

```
CREATE or REPLACE VIEW gp_toolkit.__gp_user_tables
AS
    SELECT
        fn.fnnspname as autnspname,
        fn.fnrelname as autrelname,
```

```
        relkind as autrelkind,
        reltuples as autreltuples,
        relpages as autrelpages,
        relacl as autrelacl,
        pgc.oid as autoid,
        pgc.reltoastrelid as auttoastoid,
        pgc.relstorage as autrelstorage
    FROM
        pg_catalog.pg_class pgc,
        gp_toolkit.__gp_fullname fn
    WHERE pgc.relnamespace IN
    (
        SELECT aunoid
        FROM gp_toolkit.__gp_user_namespaces
    )
    AND (pgc.relkind = 'r' OR pgc.relkind = 'm')
    AND pgc.relispopulated = 't'
    AND pgc.oid = fn.fnoid;

GRANT SELECT ON TABLE gp_toolkit.__gp_user_tables TO public;
```

The gpperfmon Database

The gpperfmon database is a dedicated database where data collection agents on Greenplum segment hosts save query and system statistics.

The gpperfmon database is created using the gpperfmon_install command-line utility. The utility creates the database and the gpmon database role and enables the data collection agents on the master and segment hosts. See the gpperfmon_install reference in the *Greenplum Database Utility Guide* for information about using the utility and configuring the data collection agents.

The gpperfmon database consists of three sets of tables that capture query and system status information at different stages.

- _now tables store current system metrics such as active queries.
- _tail tables are used to stage data before it is saved to the _history tables. The _tail tables are for internal use only and not to be queried by users.
- _history tables store historical metrics.

The data for _now and _tail tables are stored as text files on the master host file system, and are accessed in the gpperfmon database via external tables. The history tables are regular heap database tables in the gpperfmon database. History is saved only for queries that run for a minimum number of seconds, 20 by default. You can set this threshold to another value by setting the min_query_time parameter in the $MASTER_DATA_DIRECTORY/gpperfmon/conf/gpperfmon.conf configuration file. Setting the value to 0 saves history for all queries.

> **Note:** gpperfmon does not support SQL ALTER commands. ALTER queries are not recorded in the gpperfmon query history tables.

The history tables are partitioned by month. See *History Table Partition Retention* for information about removing old partitions.

The database contains the following categories of tables:

- The *database_** tables store query workload information for a Greenplum Database instance.
- The *diskspace_** tables store diskspace metrics.
- The *log_alert_** tables store error and warning messages from pg_log.
- The *queries_** tables store high-level query status information.
- The *segment_** tables store memory allocation statistics for the Greenplum Database segment instances.
- The *socket_stats_** tables store statistical metrics about socket usage for a Greenplum Database instance. Note: These tables are in place for future use and are not currently populated.
- The *system_** tables store system utilization metrics.

The gpperfmon database also contains the following views:

- The *dynamic_memory_info* view shows an aggregate of all the segments per host and the amount of dynamic memory used per host.
- The *memory_info* view shows per-host memory information from the system_history and segment_history tables.

History Table Partition Retention

The history tables in the gpperfmon database are partitioned by month. Partitions are automatically added in two month increments as needed.

The partition_age parameter in the $MASTER_DATA_DIRECTORY/gpperfmon/conf/gpperfmon.conf file can be set to the maximum number of monthly partitions to keep. Partitions older than the specified value are removed automatically when new partitions are added.

The default value for `partition_age` is `0`, which means that administrators must manually remove unneeded partitions.

Alert Log Processing and Log Rotation

When the `gp_enable_gpperfmon` server configuration parameter is set to true, the Greenplum Database syslogger writes alert messages to a `.csv` file in the `$MASTER_DATA_DIRECTORY/gpperfmon/logs` directory.

The level of messages written to the log can be set to `none`, `warning`, `error`, `fatal`, or `panic` by setting the `gpperfmon_log_alert_level` server configuration parameter in `postgresql.conf`. The default message level is `warning`.

The directory where the log is written can be changed by setting the `log_location` configuration variable in the `$MASTER_DATA_DIRECTORY/gpperfmon/conf/gpperfmon.conf` configuration file.

The syslogger rotates the alert log every 24 hours or when the current log file reaches or exceeds 1MB.

A rotated log file can exceed 1MB if a single error message contains a large SQL statement or a large stack trace. Also, the syslogger processes error messages in chunks, with a separate chunk for each logging process. The size of a chunk is OS-dependent; on Red Hat Enterprise Linux, for example, it is 4096 bytes. If many Greenplum Database sessions generate error messages at the same time, the log file can grow significantly before its size is checked and log rotation is triggered.

gpperfmon Data Collection Process

When Greenplum Database starts up with gpperfmon support enabled, it forks a `gpmmon` agent process. `gpmmon` then starts a `gpsmon` agent process on the master host and every segment host in the Greenplum Database cluster. The Greenplum Database postmaster process monitors the `gpmmon` process and restarts it if needed, and the `gpmmon` process monitors and restarts `gpsmon` processes as needed.

The `gpmmon` process runs in a loop and at configurable intervals retrieves data accumulated by the `gpsmon` processes, adds it to the data files for the `_now` and `_tail` external database tables, and then into the `_history` regular heap database tables.

> **Note:** The `log_alert` tables in the `gpperfmon` database follow a different process, since alert messages are delivered by the Greenplum Database system logger instead of through `gpsmon`. See *Alert Log Processing and Log Rotation* for more information.

Two configuration parameters in the `$MASTER_DATA_DIRECTORY/gpperfmon/conf/gpperfmon.conf` configuration file control how often `gpmmon` activities are triggered:

- The `quantum` parameter is how frequently, in seconds, `gpmmon` requests data from the `gpsmon` agents on the segment hosts and adds retrieved data to the `_now` and `_tail` external table data files. Valid values for the `quantum` parameter are 10, 15, 20, 30, and 60. The default is 15.
- The `harvest_interval` parameter is how frequently, in seconds, data in the `_tail` tables is moved to the `_history` tables. The `harvest_interval` must be at least 30. The default is 120.

See the `gpperfmon_install` management utility reference in the *Greenplum Database Utility Guide* for the complete list of gpperfmon configuration parameters.

The following steps describe the flow of data from Greenplum Database into the `gpperfmon` database when gpperfmon support is enabled.

1. While executing queries, the Greenplum Database query dispatcher and query executor processes send out query status messages in UDP datagrams. The `gp_gpperfmon_send_interval` server configuration variable determines how frequently the database sends these messages. The default is every second.
2. The `gpsmon` process on each host receives the UDP packets, consolidates and summarizes the data they contain, and adds additional host metrics, such as CPU and memory usage.
3. The `gpsmon` processes continue to accumulate data until they receive a dump command from `gpmmon`.

4. The gpsmon processes respond to a dump command by sending their accumulated status data and log alerts to a listening gpmmon event handler thread.
5. The gpmmon event handler saves the metrics to .txt files in the $MASTER_DATA_DIRECTORY/gpperfmon/data directory on the master host.

At each quantum interval (15 seconds by default), gpmmon performs the following steps:

1. Sends a dump command to the gpsmon processes.
2. Gathers and converts the .txt files saved in the $MASTER_DATA_DIRECTORY/gpperfmon/data directory into .dat external data files for the _now and _tail external tables in the gpperfmon database.

   For example, disk space metrics are added to the diskspace_now.dat and _diskspace_tail.dat delimited text files. These text files are accessed via the diskspace_now and _diskspace_tail tables in the gpperfmon database.

At each harvest_interval (120 seconds by default), gpmmon performs the following steps for each _tail file:

1. Renames the _tail file to a _stage file.
2. Creates a new _tail file.
3. Appends data from the _stage file into the _tail file.
4. Runs a SQL command to insert the data from the _tail external table into the corresponding _history table.

   For example, the contents of the _database_tail external table is inserted into the database_history regular (heap) table.
5. Deletes the _tail file after its contents have been loaded into the database table.
6. Gathers all of the gpdb-alert-*.csv files in the $MASTER_DATA_DIRECTORY/gpperfmon/logs directory (except the most recent, which the syslogger has open and is writing to) into a single file, alert_log_stage.
7. Loads the alert_log_stage file into the log_alert_history table in the gpperfmon database.
8. Truncates the alert_log_stage file.

The following topics describe the contents of the tables in the gpperfmon database.

*database_**

The database_* tables store query workload information for a Greenplum Database instance. There are three database tables, all having the same columns:

- database_now is an external table whose data files are stored in $MASTER_DATA_DIRECTORY/gpperfmon/data. Current query workload data is stored in database_now during the period between data collection from the data collection agents and automatic commitment to the database_history table.
- database_tail is an external table whose data files are stored in $MASTER_DATA_DIRECTORY/gpperfmon/data. This is a transitional table for query workload data that has been cleared from database_now but has not yet been committed to database_history. It typically only contains a few minutes worth of data.
- database_history is a regular table that stores historical database-wide query workload data. It is pre-partitioned into monthly partitions. Partitions are automatically added in two month increments as needed.

Column	Type	Description
ctime	timestamp	Time this row was created.

Column	Type	Description
queries_total	int	The total number of queries in Greenplum Database at data collection time.
queries_running	int	The number of active queries running at data collection time.
queries_queued	int	The number of queries waiting in a resource group or resource queue, depending upon which resource management scheme is active, at data collection time.

diskspace_*

The diskspace_* tables store diskspace metrics.

- diskspace_now is an external table whose data files are stored in $MASTER_DATA_DIRECTORY/gpperfmon/data. Current diskspace metrics are stored in database_now during the period between data collection from the gpperfmon agents and automatic commitment to the diskspace_history table.
- diskspace_tail is an external table whose data files are stored in $MASTER_DATA_DIRECTORY/gpperfmon/data. This is a transitional table for diskspace metrics that have been cleared from diskspace_now but has not yet been committed to diskspace_history. It typically only contains a few minutes worth of data.
- diskspace_history is a regular table that stores historical diskspace metrics. It is pre-partitioned into monthly partitions. Partitions are automatically added in two month increments as needed.

Column	Type	Description
ctime	timestamp(0) without time zone	Time of diskspace measurement.
hostname	varchar(64)	The hostname associated with the diskspace measurement.
Filesystem	text	Name of the filesystem for the diskspace measurement.
total_bytes	bigint	Total bytes in the file system.
bytes_used	bigint	Total bytes used in the file system.
bytes_available	bigint	Total bytes available in file system.

interface_stats_*

The interface_stats_* tables store statistical metrics about communications over each active interface for a Greenplum Database instance.

These tables are in place for future use and are not currently populated.

There are three interface_stats tables, all having the same columns:

- interface_stats_now is an external table whose data files are stored in $MASTER_DATA_DIRECTORY/gpperfmon/data.
- interface_stats_tail is an external table whose data files are stored in $MASTER_DATA_DIRECTORY/gpperfmon/data. This is a transitional table for statistical interface

metrics that has been cleared from `interface_stats_now` but has not yet been committed to `interface_stats_history`. It typically only contains a few minutes worth of data.

- `interface_stats_history` is a regular table that stores statistical interface metrics. It is pre-partitioned into monthly partitions. Partitions are automatically added in one month increments as needed.

Column	Type	Description
`interface_name`	string	Name of the interface. For example: eth0, eth1, lo.
`bytes_received`	bigint	Amount of data received in bytes.
`packets_received`	bigint	Number of packets received.
`receive_errors`	bigint	Number of errors encountered while data was being received.
`receive_drops`	bigint	Number of times packets were dropped while data was being received.
`receive_fifo_errors`	bigint	Number of times FIFO (first in first out) errors were encountered while data was being received.
`receive_frame_errors`	bigint	Number of frame errors while data was being received.
`receive_compressed_packets`	int	Number of packets received in compressed format.
`receive_multicast_packets`	int	Number of multicast packets received.
`bytes_transmitted`	bigint	Amount of data transmitted in bytes.
`packets_transmitted`	bigint	Amount of data transmitted in bytes.
`packets_transmitted`	bigint	Number of packets transmitted.
`transmit_errors`	bigint	Number of errors encountered during data transmission.
`transmit_drops`	bigint	Number of times packets were dropped during data transmission.
`transmit_fifo_errors`	bigint	Number of times fifo errors were encountered during data transmission.
`transmit_collision_errors`	bigint	Number of times collision errors were encountered during data transmission.
`transmit_carrier_errors`	bigint	Number of times carrier errors were encountered during data transmission.

Column	Type	Description
transmit_compressed_packets	int	Number of packets transmitted in compressed format.

log_alert_*

The log_alert_* tables store pg_log errors and warnings.

See *Alert Log Processing and Log Rotation* for information about configuring the system logger for gpperfmon.

There are three log_alert tables, all having the same columns:

- log_alert_now is an external table whose data is stored in .csv files in the $MASTER_DATA_DIRECTORY/gpperfmon/logs directory. Current pg_log errors and warnings data are available in log_alert_now during the period between data collection from the gpperfmon agents and automatic commitment to the log_alert_history table.
- log_alert_tail is an external table with data stored in $MASTER_DATA_DIRECTORY/gpperfmon/logs/alert_log_stage. This is a transitional table for data that has been cleared from log_alert_now but has not yet been committed to log_alert_history. The table includes records from all alert logs except the most recent. It typically contains only a few minutes' worth of data.
- log_alert_history is a regular table that stores historical database-wide errors and warnings data. It is pre-partitioned into monthly partitions. Partitions are automatically added in two month increments as needed.

Column	Type	Description
logtime	timestamp with time zone	Timestamp for this log
loguser	text	User of the query
logdatabase	text	The accessed database
logpid	text	Process id
logthread	text	Thread number
loghost	text	Host name or ip address
logport	text	Port number
logsessiontime	timestamp with time zone	Session timestamp
logtransaction	integer	Transaction id
logsession	text	Session id
logcmdcount	text	Command count
logsegment	text	Segment number
logslice	text	Slice number
logdistxact	text	Distributed transaction
loglocalxact	text	Local transaction
logsubxact	text	Subtransaction
logseverity	text	Log severity
logstate	text	State
logmessage	text	Log message

Column	Type	Description
logdetail	text	Detailed message
loghint	text	Hint info
logquery	text	Executed query
logquerypos	text	Query position
logcontext	text	Context info
logdebug	text	Debug
logcursorpos	text	Cursor position
logfunction	text	Function info
logfile	text	Source code file
logline	text	Source code line
logstack	text	Stack trace

queries_*

The queries_* tables store high-level query status information.

The tmid, ssid and ccnt columns are the composite key that uniquely identifies a particular query.

There are three queries tables, all having the same columns:

- queries_now is an external table whose data files are stored in $MASTER_DATA_DIRECTORY/gpperfmon/data. Current query status is stored in queries_now during the period between data collection from the gpperfmon agents and automatic commitment to the queries_history table.
- queries_tail is an external table whose data files are stored in $MASTER_DATA_DIRECTORY/gpperfmon/data. This is a transitional table for query status data that has been cleared from queries_now but has not yet been committed to queries_history. It typically only contains a few minutes worth of data.
- queries_history is a regular table that stores historical query status data. It is pre-partitioned into monthly partitions. Partitions are automatically added in two month increments as needed.

Column	Type	Description
ctime	timestamp	Time this row was created.
tmid	int	A time identifier for a particular query. All records associated with the query will have the same tmid.
ssid	int	The session id as shown by gp_session_id. All records associated with the query will have the same ssid.
ccnt	int	The command number within this session as shown by gp_command_count. All records associated with the query will have the same ccnt.

Column	Type	Description
username	varchar(64)	Greenplum role name that issued this query.
db	varchar(64)	Name of the database queried.
cost	int	Not implemented in this release.
tsubmit	timestamp	Time the query was submitted.
tstart	timestamp	Time the query was started.
tfinish	timestamp	Time the query finished.
status	varchar(64)	Status of the query -- start, done, or abort.
rows_out	bigint	Rows out for the query.
cpu_elapsed	bigint	CPU usage by all processes across all segments executing this query (in seconds). It is the sum of the CPU usage values taken from all active primary segments in the database system. Note that the value is logged as 0 if the query runtime is shorter than the value for the quantum. This occurs even if the query runtime is greater than the value for min_query_time, and this value is lower than the value for the quantum.
cpu_currpct	float	Current CPU percent average for all processes executing this query. The percentages for all processes running on each segment are averaged, and then the average of all those values is calculated to render this metric. Current CPU percent average is always zero in historical and tail data.

Column	Type	Description
skew_cpu	float	Displays the amount of processing skew in the system for this query. Processing/CPU skew occurs when one segment performs a disproportionate amount of processing for a query. This value is the coefficient of variation in the CPU% metric across all segments for this query, multiplied by 100. For example, a value of .95 is shown as 95.
skew_rows	float	Displays the amount of row skew in the system. Row skew occurs when one segment produces a disproportionate number of rows for a query. This value is the coefficient of variation for the rows_in metric across all segments for this query, multiplied by 100. For example, a value of .95 is shown as 95.
query_hash	bigint	Not implemented in this release.
query_text	text	The SQL text of this query.
query_plan	text	Text of the query plan. Not implemented in this release.
application_name	varchar(64)	The name of the application.
rsqname	varchar(64)	If the resource queue-based resource management scheme is active, this column specifies the name of the resource queue.
rqppriority	varchar(64)	If the resource queue-based resource management scheme is active, this column specifies the priority of the query -- max, high, med, low, or min.

*segment_**

The segment_* tables contain memory allocation statistics for the Greenplum Database segment instances. This tracks the amount of memory consumed by all postgres processes of a particular segment instance, and the remaining amount of memory available to a segment as per the settings configured by the currently active resource management scheme (resource group-based or resource queue-based). See the *Greenplum Database Administrator Guide* for more information about resource management schemes.

There are three segment tables, all having the same columns:

- segment_now is an external table whose data files are stored in $MASTER_DATA_DIRECTORY/gpperfmon/data. Current memory allocation data is stored in segment_now during the

period between data collection from the `gpperfmon` agents and automatic commitment to the segment_history table.

- `segment_tail` is an external table whose data files are stored in `$MASTER_DATA_DIRECTORY/gpperfmon/data`. This is a transitional table for memory allocation data that has been cleared from `segment_now` but has not yet been committed to `segment_history`. It typically only contains a few minutes worth of data.
- `segment_history` is a regular table that stores historical memory allocation metrics. It is pre-partitioned into monthly partitions. Partitions are automatically added in two month increments as needed.

A particular segment instance is identified by its `hostname` and `dbid` (the unique segment identifier as per the `gp_segment_configuration` system catalog table).

Column	Type	Description
`ctime`	timestamp(0) (without time zone)	The time the row was created.
`dbid`	int	The segment ID (`dbid` from `gp_segment_configuration`).
`hostname`	charvar(64)	The segment hostname.
`dynamic_memory_used`	bigint	The amount of dynamic memory (in bytes) allocated to query processes running on this segment.
`dynamic_memory_available`	bigint	The amount of additional dynamic memory (in bytes) that the segment can request before reaching the limit set by the currently active resource management scheme (resource group-based or resource queue-based).

See also the views `memory_info` and `dynamic_memory_info` for aggregated memory allocation and utilization by host.

*socket_stats_**

The `socket_stats_*` tables store statistical metrics about socket usage for a Greenplum Database instance. There are three system tables, all having the same columns:

These tables are in place for future use and are not currently populated.

- `socket_stats_now` is an external table whose data files are stored in `$MASTER_DATA_DIRECTORY/gpperfmon/data`.
- `socket_stats_tail` is an external table whose data files are stored in `$MASTER_DATA_DIRECTORY/gpperfmon/data`. This is a transitional table for socket statistical metrics that has been cleared from `socket_stats_now` but has not yet been committed to `socket_stats_history`. It typically only contains a few minutes worth of data.
- `socket_stats_history` is a regular table that stores historical socket statistical metrics. It is pre-partitioned into monthly partitions. Partitions are automatically added in two month increments as needed.

Column	Type	Description
total_sockets_used	int	Total sockets used in the system.
tcp_sockets_inuse	int	Number of TCP sockets in use.
tcp_sockets_orphan	int	Number of TCP sockets orphaned.
tcp_sockets_timewait	int	Number of TCP sockets in Time-Wait.
tcp_sockets_alloc	int	Number of TCP sockets allocated.
tcp_sockets_memusage_inbytes	int	Amount of memory consumed by TCP sockets.
udp_sockets_inuse	int	Number of UDP sockets in use.
udp_sockets_memusage_inbytes	int	Amount of memory consumed by UDP sockets.
raw_sockets_inuse	int	Number of RAW sockets in use.
frag_sockets_inuse	int	Number of FRAG sockets in use.
frag_sockets_memusage_inbytes	int	Amount of memory consumed by FRAG sockets.

*system_**

The system_* tables store system utilization metrics. There are three system tables, all having the same columns:

- system_now is an external table whose data files are stored in $MASTER_DATA_DIRECTORY/gpperfmon/data. Current system utilization data is stored in system_now during the period between data collection from the gpperfmon agents and automatic commitment to the system_history table.
- system_tail is an external table whose data files are stored in $MASTER_DATA_DIRECTORY/gpperfmon/data. This is a transitional table for system utilization data that has been cleared from system_now but has not yet been committed to system_history. It typically only contains a few minutes worth of data.
- system_history is a regular table that stores historical system utilization metrics. It is pre-partitioned into monthly partitions. Partitions are automatically added in two month increments as needed.

Column	Type	Description
ctime	timestamp	Time this row was created.
hostname	varchar(64)	Segment or master hostname associated with these system metrics.
mem_total	bigint	Total system memory in Bytes for this host.
mem_used	bigint	Used system memory in Bytes for this host.

Column	Type	Description
mem_actual_used	bigint	Used actual memory in Bytes for this host (not including the memory reserved for cache and buffers).
mem_actual_free	bigint	Free actual memory in Bytes for this host (not including the memory reserved for cache and buffers).
swap_total	bigint	Total swap space in Bytes for this host.
swap_used	bigint	Used swap space in Bytes for this host.
swap_page_in	bigint	Number of swap pages in.
swap_page_out	bigint	Number of swap pages out.
cpu_user	float	CPU usage by the Greenplum system user.
cpu_sys	float	CPU usage for this host.
cpu_idle	float	Idle CPU capacity at metric collection time.
load0	float	CPU load average for the prior one-minute period.
load1	float	CPU load average for the prior five-minute period.
load2	float	CPU load average for the prior fifteen-minute period.
quantum	int	Interval between metric collection for this metric entry.
disk_ro_rate	bigint	Disk read operations per second.
disk_wo_rate	bigint	Disk write operations per second.
disk_rb_rate	bigint	Bytes per second for disk read operations.
disk_wb_rate	bigint	Bytes per second for disk write operations.
net_rp_rate	bigint	Packets per second on the system network for read operations.
net_wp_rate	bigint	Packets per second on the system network for write operations.
net_rb_rate	bigint	Bytes per second on the system network for read operations.

Column	Type	Description
net_wb_rate	bigint	Bytes per second on the system network for write operations.

dynamic_memory_info

The dynamic_memory_info view shows a sum of the used and available dynamic memory for all segment instances on a segment host. Dynamic memory refers to the maximum amount of memory that Greenplum Database instance will allow the query processes of a single segment instance to consume before it starts cancelling processes. This limit, determined by the currently active resource management scheme (resource group-based or resource queue-based), is evaluated on a per-segment basis.

Column	Type	Description
ctime	timestamp(0) without time zone	Time this row was created in the segment_history table.
hostname	varchar(64)	Segment or master hostname associated with these system memory metrics.
dynamic_memory_used_mb	numeric	The amount of dynamic memory in MB allocated to query processes running on this segment.
dynamic_memory_available_mb	numeric	The amount of additional dynamic memory (in MB) available to the query processes running on this segment host. Note that this value is a sum of the available memory for all segments on a host. Even though this value reports available memory, it is possible that one or more segments on the host have exceeded their memory limit.

memory_info

The memory_info view shows per-host memory information from the system_history and segment_history tables. This allows administrators to compare the total memory available on a segment host, total memory used on a segment host, and dynamic memory used by query processes.

Column	Type	Description
ctime	timestamp(0) without time zone	Time this row was created in the segment_history table.
hostname	varchar(64)	Segment or master hostname associated with these system memory metrics.
mem_total_mb	numeric	Total system memory in MB for this segment host.
mem_used_mb	numeric	Total system memory used in MB for this segment host.

Column	Type	Description
mem_actual_used_mb	numeric	Actual system memory used in MB for this segment host.
mem_actual_free_mb	numeric	Actual system memory free in MB for this segment host.
swap_total_mb	numeric	Total swap space in MB for this segment host.
swap_used_mb	numeric	Total swap space used in MB for this segment host.
dynamic_memory_used_mb	numeric	The amount of dynamic memory in MB allocated to query processes running on this segment.
dynamic_memory_available_mb	numeric	The amount of additional dynamic memory (in MB) available to the query processes running on this segment host. Note that this value is a sum of the available memory for all segments on a host. Even though this value reports available memory, it is possible that one or more segments on the host have exceeded their memory limit.

Server Programmatic Interfaces

This section describes programmatic interfaces to the Greenplum Database server.

Greenplum Partner Connector API

With the Greenplum Partner Connector API (GPPC API), you can write portable Greenplum Database user-defined functions (UDFs) in the C and C++ programming languages. Functions that you develop with the GPPC API require no recompilation or modification to work with older or newer Greenplum Database versions.

Functions that you write to the GPPC API can be invoked using SQL in Greenplum Database. The API provides a set of functions and macros that you can use to issue SQL commands through the Server Programming Interface (SPI), manipulate simple and composite data type function arguments and return values, manage memory, and handle data.

You compile the C/C++ functions that you develop with the GPPC API into a shared library. The GPPC functions are available to Greenplum Database users after the shared library is installed in the Greenplum Database cluster and the GPPC functions are registered as SQL UDFs.

> **Note:** The Greenplum Partner Connector is supported for Greenplum Database versions 4.3.5.0 and later.

This topic contains the following information:

Using the GPPC API

The GPPC API shares some concepts with C language functions as defined by PostgreSQL. Refer to *C-Language Functions* in the PostgreSQL documentation for detailed information about developing C language functions.

The GPPC API is a wrapper that makes a C/C++ function SQL-invokable in Greenplum Database. This wrapper shields GPPC functions that you write from Greenplum Database library changes by normalizing table and data manipulation and SPI operations through functions and macros defined by the API.

The GPPC API includes functions and macros to:

- Operate on base and composite data types.
- Process function arguments and return values.
- Allocate and free memory.
- Log and report errors to the client.
- Issue SPI queries.
- Return a table or set of rows.
- Process tables as function input arguments.

Requirements

When you develop with the GPPC API:

- You must develop your code on a system with the same hardware and software architecture as that of your Greenplum Database hosts.
- You must write the GPPC function(s) in the C or C++ programming languages.
- The function code must use the GPPC API, data types, and macros.
- The function code must *not* use the PostgreSQL C-Language Function API, header files, functions, or macros.
- The function code must *not* `#include` the `postgres.h` header file or use `PG_MODULE_MAGIC`.
- You must use only the GPPC-wrapped memory functions to allocate and free memory. See *Memory Handling*.
- Symbol names in your object files must not conflict with each other nor with symbols defined in the Greenplum Database server. You must rename your functions or variables if you get error messages to this effect.

Header and Library Files

The GPPC header files and libraries are installed in `$GPHOME`:

- $GPHOME/include/gppc.h - the main GPPC header file
- $GPHOME/include/gppc_config.h - header file defining the GPPC version
- $GPHOME/lib/libgppc.[a, so, so.1, so.1.2] - GPPC archive and shared libraries

Data Types

The GPPC functions that you create will operate on data residing in Greenplum Database. The GPPC API includes data type definitions for equivalent Greenplum Database SQL data types. You must use these types in your GPPC functions.

The GPPC API defines a generic data type that you can use to represent any GPPC type. This data type is named `GppcDatum`, and is defined as follows:

```
typedef int64_t GppcDatum;
```

The following table identifies each GPPC data type and the SQL type to which it maps.

SQL Type	GPPC Type	GPPC Oid for Type
boolean	GppcBool	GppcOidBool
char (single byte)	GppcChar	GppcOidChar
int2/smallint	GppcInt2	GppcOidInt2

SQL Type	GPPC Type	GPPC Oid for Type
int4/integer	GppcInt4	GppcOidInt4
int8/bigint	GppcInt8	GppcOidInt8
float4/real	GppcFloat4	GppcOidFloat4
float8/double	GppcFloat8	GppcOidFloat8
text	*GppcText	GppcOidText
varchar	*GppcVarChar	GppcOidVarChar
char	*GppcBpChar	GppcOidBpChar
bytea	*GppcBytea	GppcOidBytea
numeric	*GppcNumeric	GppcOidNumeric
date	GppcDate	GppcOidDate
time	GppcTime	GppcOidTime
timetz	*GppcTimeTz	GppcOidTimeTz
timestamp	GppcTimestamp	GppcOidTimestamp
timestamptz	GppcTimestampTz	GppcOidTimestampTz
anytable	GppcAnyTable	GppcOidAnyTable
oid	GppcOid	

The GPPC API treats text, numeric, and timestamp data types specially, providing functions to operate on these types.

Example GPPC base data type declarations:

```
GppcText       message;
GppcInt4       arg1;
GppcNumeric    total_sales;
```

The GPPC API defines functions to convert between the generic `GppcDatum` type and the GPPC specific types. For example, to convert from an integer to a datum:

```
GppcInt4 num = 13;
GppcDatum num_dat = GppcInt4GetDatum(num);
```

Composite Types

A composite data type represents the structure of a row or record, and is comprised of a list of field names and their data types. This structure information is typically referred to as a tuple descriptor. An instance of a composite type is typically referred to as a tuple or row. A tuple does not have a fixed layout and can contain null fields.

The GPPC API provides an interface that you can use to define the structure of, to access, and to set tuples. You will use this interface when your GPPC function takes a table as an input argument or returns table or set of record types. Using tuples in table and set returning functions is covered later in this topic.

Function Declaration, Arguments, and Results

The GPPC API relies on macros to declare functions and to simplify the passing of function arguments and results. These macros include:

Task	Macro Signature	Description
Make a function SQL-invokable	`GPPC_FUNCTION_INFO(function_name)`	Glue to make function *`function_name`* SQL-invokable.
Declare a function	`GppcDatum function_name(GPPC_FUNCTION_ARGS)`	Declare a GPPC function named *`function_name`*; every function must have this same signature.
Return the number of arguments	`GPPC_NARGS()`	Return the number of arguments passed to the function.
Fetch an argument	`GPPC_GETARG_<ARGTYPE>(arg_num)`	Fetch the value of argument number *arg_num* (starts at 0), where <ARGTYPE> identifies the data type of the argument. For example, `GPPC_GETARG_FLOAT8(0)`.
Fetch and make a copy of a text-type argument	`GPPC_GETARG_<ARGTYPE>_COPY(arg_num)`	Fetch and make a copy of the value of argument number *arg_num* (starts at 0). <ARGTYPE> identifies the text type (text, varchar, bpchar, bytea). For example, `GPPC_GETARG_BYTEA_COPY(1)`.
Determine if an argument is NULL	`GPPC_ARGISNULL(arg_num)`	Return whether or not argument number *`arg_num`* is NULL.
Return a result	`GPPC_RETURN_<ARGTYPE>(return_val)`	Return the value *`return_val`*, where <ARGTYPE> identifies the data type of the return value. For example, `GPPC_RETURN_INT4(131)`.

When you define and implement your GPPC function, you must declare it with the GPPC API using the two declarations identified above. For example, to declare a GPPC function named `add_int4s()`:

```
GPPC_FUNCTION_INFO(add_int4s);
GppcDatum add_int4s(GPPC_FUNCTION_ARGS);

GppcDatum
add_int4s(GPPC_FUNCTION_ARGS)
{
  // code here
}
```

If the `add_int4s()` function takes two input arguments of type `int4`, you use the `GPPC_GETARG_INT4(arg_num)` macro to access the argument values. The argument index starts at 0. For example:

```
GppcInt4  first_int = GPPC_GETARG_INT4(0);
GppcInt4  second_int = GPPC_GETARG_INT4(1);
```

If `add_int4s()` returns the sum of the two input arguments, you use the `GPPC_RETURN_INT8(return_val)` macro to return this sum. For example:

```
GppcInt8  sum = first_int + second_int;
```

```
GPPC_RETURN_INT8(sum);
```

The complete GPPC function:

```
GPPC_FUNCTION_INFO(add_int4s);
GppcDatum add_int4s(GPPC_FUNCTION_ARGS);

GppcDatum
add_int4s(GPPC_FUNCTION_ARGS)
{
  // get input arguments
  GppcInt4    first_int = GPPC_GETARG_INT4(0);
  GppcInt4    second_int = GPPC_GETARG_INT4(1);

  // add the arguments
  GppcInt8    sum = first_int + second_int;

  // return the sum
  GPPC_RETURN_INT8(sum);
}
```

Memory Handling

The GPPC API provides functions that you use to allocate and free memory, including text memory. You must use these functions for all memory operations.

Function Name	Description
void *GppcAlloc(size_t *num*)	Allocate *num* bytes of uninitialized memory.
void *GppcAlloc0(size_t *num*)	Allocate *num* bytes of 0-initialized memory.
void *GppcRealloc(void **ptr*, size_t *num*)	Resize pre-allocated memory.
void GppcFree(void **ptr*)	Free allocated memory.

After you allocate memory, you can use system functions such as `memcpy()` to set the data.

The following example allocates an array of `GppcDatums` and sets the array to datum versions of the function input arguments:

```
GppcDatum  *values;
int attnum = GPPC_NARGS();

// allocate memory for attnum values
values = GppcAlloc( sizeof(GppcDatum) * attnum );

// set the values
for( int i=0; i<attnum; i++ ) {
    GppcDatum d = GPPC_GETARG_DATUM(i);
    values[i] = d;
}
```

When you allocate memory for a GPPC function, you allocate it in the current context. The GPPC API includes functions to return, create, switch, and reset memory contexts.

Function Name	Description
GppcMemoryContext GppcGetCurrentMemoryContext(void)	Return the current memory context.
GppcMemoryContext GppcMemoryContextCreate(GppcMemoryContext *parent*)	Create a new memory context under *parent*.

Function Name	Description
GppcMemoryContext GppcMemoryContextSwitchTo(GppcMemoryContext *context*)	Switch to the memory context *context*.
void GppcMemoryContextReset(GppcMemoryContext *context*)	Reset (free) the memory in memory context *context*.

Greenplum Database typically calls a SQL-invoked function in a per-tuple context that it creates and deletes every time the server backend processes a table row. Do not assume that memory allocated in the current memory context is available across multiple function calls.

Working With Variable-Length Text Types

The GPPC API supports the variable length text, varchar, blank padded, and byte array types. You must use the GPPC API-provided functions when you operate on these data types. Variable text manipulation functions provided in the GPPC API include those to allocate memory for, determine string length of, get string pointers for, and access these types:

Function Name	Description
GppcText GppcAllocText(size_t *len*) GppcVarChar GppcAllocVarChar(size_t *len*) GppcBpChar GppcAllocBpChar(size_t *len*) GppcBytea GppcAllocBytea(size_t *len*)	Allocate *len* bytes of memory for the varying length type.
size_t GppcGetTextLength(GppcText *s*) size_t GppcGetVarCharLength(GppcVarChar *s*) size_t GppcGetBpCharLength(GppcBpChar *s*) size_t GppcGetByteaLength(GppcBytea *b*)	Return the number of bytes in the memory chunk.
char *GppcGetTextPointer(GppcText *s*) char *GppcGetVarCharPointer(GppcVarChar *s*) char *GppcGetBpCharPointer(GppcBpChar *s*) char *GppcGetByteaPointer(GppcBytea *b*)	Return a string pointer to the head of the memory chunk. The string is not null-terminated.
char *GppcTextGetCString(GppcText *s*) char *GppcVarCharGetCString(GppcVarChar *s*) char *GppcBpCharGetCString(GppcBpChar *s*)	Return a string pointer to the head of the memory chunk. The string is null-terminated.
GppcText *GppcCStringGetText(const char **s*) GppcVarChar *GppcCStringGetVarChar(const char **s*) GppcBpChar *GppcCStringGetBpChar(const char **s*)	Build a varying-length type from a character string.

Memory returned by the `GppcGet<VLEN_ARGTYPE>Pointer()` functions may point to actual database content. Do not modify the memory content. The GPPC API provides functions to allocate memory for these types should you require it. After you allocate memory, you can use system functions such as `memcpy()` to set the data.

The following example manipulates text input arguments and allocates and sets result memory for a text string concatenation operation:

```
GppcText first_textstr = GPPC_GETARG_TEXT(0);
GppcText second_textstr = GPPC_GETARG_TEXT(1);

// determine the size of the concatenated string and allocate
// text memory of this size
size_t arg0_len = GppcGetTextLength(first_textstr);
size_t arg1_len = GppcGetTextLength(second_textstr);
GppcText retstring = GppcAllocText(arg0_len + arg1_len);

// construct the concatenated return string; copying each string
// individually
memcpy(GppcGetTextPointer(retstring), GppcGetTextPointer(first_textstr),
 arg0_len);
memcpy(GppcGetTextPointer(retstring) + arg0_len,
 GppcGetTextPointer(second_textstr), arg1_len);
```

Error Reporting and Logging

The GPPC API provides error reporting and logging functions. The API defines reporting levels equivalent to those in Greenplum Database:

```
typedef enum GppcReportLevel
{
        GPPC_DEBUG1                             = 10,
        GPPC_DEBUG2                             = 11,
        GPPC_DEBUG3                             = 12,
        GPPC_DEBUG4                             = 13,
        GPPC_DEBUG                              = 14,
        GPPC_LOG                                = 15,
        GPPC_INFO                               = 17,
        GPPC_NOTICE                             = 18,
        GPPC_WARNING                    = 19,
        GPPC_ERROR                              = 20,
} GppcReportLevel;
```

(The Greenplum Database *client_min_messages* server configuration parameter governs the current client logging level. The *log_min_messages* configuration parameter governs the current log-to-logfile level.)

A GPPC report includes the report level, a report message, and an optional report callback function.

Reporting and handling functions provide by the GPPC API include:

Function Name	Description
GppcReport()	Format and print/log a string of the specified report level.
GppcInstallReportCallback()	Register/install a report callback function.
GppcUninstallReportCallback()	Uninstall a report callback function.
GppcGetReportLevel()	Retrieve the level from an error report.
GppcGetReportMessage()	Retrieve the message from an error report.
GppcCheckForInterrupts()	Error out if an interrupt is pending.

The `GppcReport()` function signature is:

```
void GppcReport(GppcReportLevel elevel, const char *fmt, ...);
```

`GppcReport()` takes a format string input argument similar to `printf()`. The following example generates an error level report message that formats a GPPC text argument:

```
GppcText  uname = GPPC_GETARG_TEXT(1);
GppcReport(GPPC_ERROR, "Unknown user name: %s", GppcTextGetCString(uname));
```

Refer to the *GPPC example code* for example report callback handlers.

SPI Functions

The Greenplum Database Server Programming Interface (SPI) provides writers of C/C++ functions the ability to run SQL commands within a GPPC function. For additional information on SPI functions, refer to *Server Programming Interface* in the PostgreSQL documentation.

The GPPC API exposes a subset of PostgreSQL SPI functions. This subset enables you to issue SPI queries and retrieve SPI result values in your GPPC function. The GPPC SPI wrapper functions are:

SPI Function Name	Description
SPI_connect()	GppcSPIConnect() Connect to the Greenplum Database server programming interface.
SPI_finish()	GppcSPIFinish() Disconnect from the Greenplum Database server programming interface.
SPI_exec()	GppcSPIExec() Execute a SQL statement, returning the number of rows.
SPI_getvalue()	GppcSPIGetValue() Return the value of a specific attribute by number from a SQL result as a character string.
	GppcSPIGetDatum() Return the value of a specific attribute by number from a SQL result as a `GppcDatum`.
	GppcSPIGetValueByName() Return the value of a specific attribute by name from a SQL result as a character string.
	GppcSPIGetDatumByName() Return the value of a specific attribute by name from a SQL result as a `GppcDatum`.

When you create a GPPC function that accesses the server programming interface, your function should comply with the following flow:

```
GppcSPIConnect();
GppcSPIExec(...)
// process the results - GppcSPIGetValue(...), GppcSPIGetDatum(...)
GppcSPIFinish()
```

You use `GppcSPIExec()` to execute SQL statements in your GPPC function. When you call this function, you also identify the maximum number of rows to return. The function signature of `GppcSPIExec()` is:

```
GppcSPIResult GppcSPIExec(const char *sql_statement, long rcount);
```

`GppcSPIExec()` returns a `GppcSPIResult` structure. This structure represents SPI result data. It includes a pointer to the data, information about the number of rows processed, a counter, and a result code. The GPPC API defines this structure as follows:

```
typedef struct GppcSPIResultData
{
    struct GppcSPITupleTableData   *tuptable;
    uint32_t                       processed;
    uint32_t                       current;
    int                            rescode;
} GppcSPIResultData;
typedef GppcSPIResultData *GppcSPIResult;
```

You can set and use the `current` field in the `GppcSPIResult` structure to examine each row of the `tuptable` result data.

The following code excerpt uses the GPPC API to connect to SPI, execute a simple query, loop through query results, and finish processing:

```
GppcSPIResult   result;
char            *attname = "id";
char            *query = "SELECT i, 'foo' || i AS val FROM
 generate_series(1, 10)i ORDER BY 1";
bool            isnull = true;

// connect to SPI
if( GppcSPIConnect() < 0 ) {
    GppcReport(GPPC_ERROR, "cannot connect to SPI");
}

// execute the query, returning all rows
result = GppcSPIExec(query, 0);

// process result
while( result->current < result->processed ) {
    // get the value of attname column as a datum, making a copy
    datum = GppcSPIGetDatumByName(result, attname, &isnull, true);

    // do something with value

    // move on to next row
    result->current++;
}

// complete processing
GppcSPIFinish();
```

About Tuple Descriptors and Tuples

A table or a set of records contains one or more tuples (rows). The structure of each attribute of a tuple is defined by a tuple descriptor. A tuple descriptor defines the following for each attribute in the tuple:

- attribute name
- object identifier of the attribute data type
- byte length of the attribute data type
- object identifier of the attribute modifer

The GPPC API defines an abstract type, `GppcTupleDesc`, to represent a tuple/row descriptor. The API also provides functions that you can use to create, access, and set tuple descriptors:

Function Name	Description
GppcCreateTemplateTupleDesc()	Create an empty tuple descriptor with a specified number of attributes.
GppcTupleDescInitEntry()	Add an attribute to the tuple descriptor at a specified position.
GppcTupleDescNattrs()	Fetch the number of attributes in the tuple descriptor.
GppcTupleDescAttrName()	Fetch the name of the attribute in a specific position (starts at 0) in the tuple descriptor.
GppcTupleDescAttrType()	Fetch the type object identifier of the attribute in a specific position (starts at 0) in the tuple descriptor.
GppcTupleDescAttrLen()	Fetch the type length of an attribute in a specific position (starts at 0) in the tuple descriptor.
GppcTupleDescAttrTypmod()	Fetch the type modifier object identifier of an attribute in a specific position (starts at 0) in the tuple descriptor.

To construct a tuple descriptor, you first create a template, and then fill in the descriptor fields for each attribute. The signatures for these functions are:

```
GppcTupleDesc GppcCreateTemplateTupleDesc(int natts);
void GppcTupleDescInitEntry(GppcTupleDesc desc, uint16_t attno,
                            const char *attname, GppcOid typid, int32_t
 typmod);
```

In some cases, you may want to initialize a tuple descriptor entry from an attribute definition in an existing tuple. The following functions fetch the number of attributes in a tuple descriptor, as well as the definition of a specific attribute (by number) in the descriptor:

```
int GppcTupleDescNattrs(GppcTupleDesc tupdesc);
const char *GppcTupleDescAttrName(GppcTupleDesc tupdesc, int16_t attno);
GppcOid GppcTupleDescAttrType(GppcTupleDesc tupdesc, int16_t attno);
int16_t GppcTupleDescAttrLen(GppcTupleDesc tupdesc, int16_t attno);
int32_t GppcTupleDescAttrTypmod(GppcTupleDesc tupdesc, int16_t attno);
```

The following example initializes a two attribute tuple descriptor. The first attribute is initialized with the definition of an attribute from a different descriptor, and the second attribute is initialized to a boolean type attribute:

```
GppcTupleDesc       tdesc;
GppcTupleDesc       indesc = some_input_descriptor;

// initialize the tuple descriptor with 2 attributes
tdesc = GppcCreateTemplateTupleDesc(2);

// use third attribute from the input descriptor
GppcTupleDescInitEntry(tdesc, 1,
        GppcTupleDescAttrName(indesc, 2),
        GppcTupleDescAttrType(indesc, 2),
        GppcTupleDescAttrTypmod(indesc, 2));

// create the boolean attribute
GppcTupleDescInitEntry(tdesc, 2, "is_active", GppcOidBool, 0);
```

The GPPC API defines an abstract type, `GppcHeapTuple`, to represent a tuple/record/row. A tuple is defined by its tuple descriptor, the value for each tuple attribute, and an indicator of whether or not each value is NULL.

The GPPC API provides functions that you can use to set and access a tuple and its attributes:

Function Name	Description
GppcHeapFormTuple()	Form a tuple from an array of `GppcDatum`s.
GppcBuildHeapTupleDatum()	Form a `GppcDatum` tuple from an array of `GppcDatum`s.
GppcGetAttributeByName()	Fetch an attribute from the tuple by name.
GppcGetAttributeByNum()	Fetch an attribute from the tuple by number (starts at 1).

The signatures for the tuple-building GPPC functions are:

```
GppcHeapTuple GppcHeapFormTuple(GppcTupleDesc tupdesc, GppcDatum *values,
 bool *nulls);
GppcDatum    GppcBuildHeapTupleDatum(GppcTupleDesc tupdesc, GppcDatum
 *values, bool *nulls);
```

The following code excerpt constructs a `GppcDatum` tuple from the tuple descriptor in the above code example, and from integer and boolean input arguments to a function:

```
GppcDatum intarg = GPPC_GETARG_INT4(0);
GppcDatum boolarg = GPPC_GETARG_BOOL(1);
GppcDatum result, values[2];
bool nulls[2] = { false, false };

// construct the values array
values[0] = intarg;
values[1] = boolarg;
result = GppcBuildHeapTupleDatum( tdesc, values, nulls );
```

Set-Returning Functions

Greenplum Database UDFs whose signatures include `RETURNS SETOF RECORD` or `RETURNS TABLE( ... )` are set-returning functions.

The GPPC API provides support for returning sets (for example, multiple rows/tuples) from a GPPC function. Greenplum Database calls a set-returning function (SRF) once for each row or item. The function must save enough state to remember what it was doing and to return the next row on each call. Memory that you allocate in the SRF context must survive across multiple function calls.

The GPPC API provides macros and functions to help keep track of and set this context, and to allocate SRF memory. They include:

Function/Macro Name	Description
GPPC_SRF_RESULT_DESC()	Get the output row tuple descriptor for this SRF. The result tuple descriptor is determined by an output table definition or a `DESCRIBE` function.
GPPC_SRF_IS_FIRSTCALL()	Determine if this is the first call to the SRF.
GPPC_SRF_FIRSTCALL_INIT()	Initialize the SRF context.
GPPC_SRF_PERCALL_SETUP()	Restore the context on each call to the SRF.
GPPC_SRF_RETURN_NEXT()	Return a value from the SRF and continue processing.
GPPC_SRF_RETURN_DONE()	Signal that SRF processing is complete.
GppSRFAlloc()	Allocate memory in this SRF context.

Function/Macro Name	Description
GppSRFAlloc0()	Allocate memory in this SRF context and initialize it to zero.
GppSRFSave()	Save user state in this SRF context.
GppSRFRestore()	Restore user state in this SRF context.

The `GppcFuncCallContext` structure provides the context for an SRF. You create this context on the first call to your SRF. Your set-returning GPPC function must retrieve the function context on each invocation. For example:

```
// set function context
GppcFuncCallContext fctx;
if (GPPC_SRF_IS_FIRSTCALL()) {
    fctx = GPPC_SRF_FIRSTCALL_INIT();
}
fctx = GPPC_SRF_PERCALL_SETUP();
// process the tuple
```

The GPPC function must provide the context when it returns a tuple result or to indicate that processing is complete. For example:

```
GPPC_SRF_RETURN_NEXT(fctx, result_tuple);
// or
GPPC_SRF_RETURN_DONE(fctx);
```

Use a `DESCRIBE` function to define the output tuple descriptor of a function that uses the `RETURNS SETOF RECORD` clause. Use the `GPPC_SRF_RESULT_DESC()` macro to get the output tuple descriptor of a function that uses the `RETURNS TABLE( ... )` clause.

Refer to the *GPPC Set-Returning Function Example* for a set-returning function code and deployment example.

Table Functions

The GPPC API provides the `GppcAnyTable` type to pass a table to a function as an input argument, or to return a table as a function result.

Table-related functions and macros provided in the GPPC API include:

Function/Macro Name	Description
GPPC_GETARG_ANYTABLE()	Fetch an anytable function argument.
GPPC_RETURN_ANYTABLE()	Return the table.
GppcAnyTableGetTupleDesc()	Fetch the tuple descriptor for the table.
GppcAnyTableGetNextTuple()	Fetch the next row in the table.

You can use the `GPPC_GETARG_ANYTABLE()` macro to retrieve a table input argument. When you have access to the table, you can examine the tuple descriptor for the table using the `GppcAnyTableGetTupleDesc()` function. The signature of this function is:

```
GppcTupleDesc GppcAnyTableGetTupleDesc(GppcAnyTable t);
```

For example, to retrieve the tuple descriptor of a table that is the first input argument to a function:

```
GppcAnyTable     intbl;
```

```
GppcTupleDesc    in_desc;

intbl = GPPC_GETARG_ANYTABLE(0);
in_desc = GppcAnyTableGetTupleDesc(intbl);
```

The `GppcAnyTableGetNextTuple()` function fetches the next row from the table. Similarly, to retrieve the next tuple from the table above:

```
GppcHeapTuple    ntuple;

ntuple = GppcAnyTableGetNextTuple(intbl);
```

Limitations

The GPPC API does not support the following operators with Greenplum Database version 5.0.x:

- integer || integer
- integer = text
- text < integer

Sample Code

The *gppc test* directory in the Greenplum Database github repository includes sample GPPC code:

- `gppc_demo/` - sample code exercising GPPC SPI functions, error reporting, data type argument and return macros, set-returning functions, and encoding functions
- `tabfunc_gppc_demo/` - sample code exercising GPPC table and set-returning functions

Building a GPPC Shared Library with PGXS

You compile functions that you write with the GPPC API into one or more shared libraries that the Greenplum Database server loads on demand.

You can use the PostgreSQL build extension infrastructure (PGXS) to build the source code for your GPPC functions against a Greenplum Database installation. This framework automates common build rules for simple modules. If you have a more complicated use case, you will need to write your own build system.

To use the PGXS infrastructure to generate a shared library for functions that you create with the GPPC API, create a simple `Makefile` that sets PGXS-specific variables.

> **Note:** Refer to *Extension Building Infrastructure* in the PostgreSQL documentation for information about the `Makefile` variables supported by PGXS.

For example, the following `Makefile` generates a shared library named `sharedlib_name.so` from two C source files named `src1.c` and `src2.c`:

```
MODULE_big = sharedlib_name
OBJS = src1.o src2.o
PG_CPPFLAGS = -I$(shell $(PG_CONFIG) --includedir)
SHLIB_LINK = -L$(shell $(PG_CONFIG) --libdir) -lgppc

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

`MODULE_big` identifes the base name of the shared library generated by the `Makefile`.

`PG_CPPFLAGS` adds the Greenplum Database installation include directory to the compiler header file search path.

`SHLIB_LINK` adds the Greenplum Database installation library directory to the linker search path. This variable also adds the GPPC library (`-lgppc`) to the link command.

The `PG_CONFIG` and `PGXS` variable settings and the `include` statement are required and typically reside in the last three lines of the `Makefile`.

Registering a GPPC Function with Greenplum Database

Before users can invoke a GPPC function from SQL, you must register the function with Greenplum Database.

Registering a GPPC function involves mapping the GPPC function signature to a SQL user-defined function. You define this mapping with the `CREATE FUNCTION .. AS` command specifying the GPPC shared library name. You may choose to use the same name or differing names for the GPPC and SQL functions.

Sample `CREATE FUNCTION ... AS` syntax follows:

```
CREATE FUNCTION sql_function_name(arg[, ...]) RETURNS return_type
  AS 'shared_library_path'[, 'gppc_function_name']
LANGUAGE C STRICT [WITH (DESCRIBE=describe_function)];
```

You may omit the shared library `.so` extension when you specify *`shared_library_path`*.

The following command registers the example `add_int4s()` function referenced earlier in this topic to a SQL UDF named `add_two_int4s_gppc()` if the GPPC function was compiled and linked in a shared library named `gppc_try.so`:

```
CREATE FUNCTION add_two_int4s_gppc(int4, int4) RETURNS int8
  AS 'gppc_try.so', 'add_int4s'
LANGUAGE C STRICT;
```

About Dynamic Loading

You specify the name of the GPPC shared library in the SQL `CREATE FUNCTION ... AS` command to register a GPPC function in the shared library with Greenplum Database. The Greenplum Database dynamic loader loads a GPPC shared library file into memory the first time that a user invokes a user-defined function linked in that shared library. If you do not provide an absolute path to the shared library in the `CREATE FUNCTION ... AS` command, Greenplum Database attempts to locate the library using these ordered steps:

1. If the shared library file path begins with the string `$libdir`, Greenplum Database looks for the file in the PostgreSQL package library directory. Run the `pg_config --pkglibdir` command to determine the location of this directory.
2. If the shared library file name is specified without a directory prefix, Greenplum Database searches for the file in the directory identified by the `dynamic_library_path` server configuration parameter value.
3. The current working directory.

Packaging and Deployment Considerations

You must package the GPPC shared library and SQL function registration script in a form suitable for deployment by the Greenplum Database administrator in the Greenplum cluster. Provide specific deployment instructions for your GPPC package.

When you construct the package and deployment instructions, take into account the following:

- Consider providing a shell script or program that the Greenplum Database administrator runs to both install the shared library to the desired file system location and register the GPPC functions.

- The GPPC shared library must be installed to the same file system location on the master host and on every segment host in the Greenplum Database cluster.
- The `gpadmin` user must have permission to traverse the complete file system path to the GPPC shared library file.
- The file system location of your GPPC shared library after it is installed in the Greenplum Database deployment determines how you reference the shared library when you register a function in the library with the `CREATE FUNCTION ... AS` command.
- Create a `.sql` script file that registers a SQL UDF for each GPPC function in your GPPC shared library. The functions that you create in the `.sql` registration script must reference the deployment location of the GPPC shared library. Include this script in your GPPC deployment package.
- Document the instructions for running your GPPC package deployment script, if you provide one.
- Document the instructions for installing the GPPC shared library if you do not include this task in a package deployment script.
- Document the instructions for installing and running the function registration script if you do not include this task in a package deployment script.

GPPC Text Function Example

In this example, you develop, build, and deploy a GPPC shared library and register and run a GPPC function named `concat_two_strings`. This function uses the GPPC API to concatenate two string arguments and return the result.

You will develop the GPPC function on your Greenplum Database master host. Deploying the GPPC shared library that you create in this example requires administrative access to your Greenplum Database cluster.

Perform the following procedure to run the example:

1. Log in to the Greenplum Database master host and set up your environment. For example:

```
$ ssh gpadmin@<gpmaster>
gpadmin@gpmaster$ . /usr/local/greenplum-db/greenplum_path.sh
```

2. Create a work directory and navigate to the new directory. For example:

```
gpadmin@gpmaster$ mkdir gppc_work
gpadmin@gpmaster$ cd gppc_work
```

3. Prepare a file for GPPC source code by opening the file in the editor of your choice. For example, to open a file named `gppc_concat.c` using `vi`:

```
gpadmin@gpmaster$ vi gppc_concat.c
```

4. Copy/paste the following code into the file:

```
#include <stdio.h>
#include <string.h>
#include "gppc.h"

// make the function SQL-invokable
GPPC_FUNCTION_INFO(concat_two_strings);

// declare the function
GppcDatum concat_two_strings(GPPC_FUNCTION_ARGS);

GppcDatum
concat_two_strings(GPPC_FUNCTION_ARGS)
{
    // retrieve the text input arguments
    GppcText arg0 = GPPC_GETARG_TEXT(0);
```

```
    GppcText arg1 = GPPC_GETARG_TEXT(1);

    // determine the size of the concatenated string and allocate
    // text memory of this size
    size_t arg0_len = GppcGetTextLength(arg0);
    size_t arg1_len = GppcGetTextLength(arg1);
    GppcText retstring = GppcAllocText(arg0_len + arg1_len);

    // construct the concatenated return string
    memcpy(GppcGetTextPointer(retstring), GppcGetTextPointer(arg0),
 arg0_len);
    memcpy(GppcGetTextPointer(retstring) + arg0_len,
 GppcGetTextPointer(arg1), arg1_len);

    GPPC_RETURN_TEXT( retstring );
}
```

The code declares and implements the `concat_two_strings()` function. It uses GPPC data types, macros, and functions to get the function arguments, allocate memory for the concatenated string, copy the arguments into the new string, and return the result.

5. Save the file and exit the editor.
6. Open a file named `Makefile` in the editor of your choice. Copy/paste the following text into the file:

```
MODULE_big = gppc_concat
OBJS = gppc_concat.o

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)

PG_CPPFLAGS = -I$(shell $(PG_CONFIG) --includedir)
SHLIB_LINK = -L$(shell $(PG_CONFIG) --libdir) -lgppc
include $(PGXS)
```

7. Save the file and exit the editor.
8. Build a GPPC shared library for the `concat_two_strings()` function. For example:

```
gpadmin@gpmaster$ make all
```

The `make` command generates a shared library file named `gppc_concat.so` in the current working directory.

9. Copy the shared library to your Greenplum Database installation. You must have Greenplum Database administrative privileges to copy the file. For example:

```
gpadmin@gpmaster$ cp gppc_concat.so /usr/local/greenplum-db/lib/
postgresql/
```

10. Copy the shared library to every host in your Greenplum Database installation. For example, if `seghostfile` contains a list, one-host-per-line, of the segment hosts in your Greenplum Database cluster:

```
gpadmin@gpmaster$ gpscp -v -f seghostfile /usr/local/greenplum-db/lib/
postgresql/gppc_concat.so =:/usr/local/greenplum-db/lib/postgresql/
gppc_concat.so
```

11. Open a `psql` session. For example:

```
gpadmin@gpmaster$ psql -d testdb
```

12. Register the GPPC function named `concat_two_strings()` with Greenplum Database, For example, to map the Greenplum Database function `concat_with_gppc()` to the GPPC `concat_two_strings()` function:

```
testdb=# CREATE FUNCTION concat_with_gppc(text, text) RETURNS text
  AS 'gppc_concat', 'concat_two_strings'
LANGUAGE C STRICT;
```

13. Run the `concat_with_gppc()` function. For example:

```
testdb=# SELECT concat_with_gppc( 'happy', 'monday' );
 concat_with_gppc
------------------
 happymonday
(1 row)
```

GPPC Set-Returning Function Example

In this example, you develop, build, and deploy a GPPC shared library. You also create and run a `.sql` registration script for a GPPC function named `return_tbl()`. This function uses the GPPC API to take an input table with an integer and a text column, determine if the integer column is greater than 13, and returns a result table with the input integer column and a boolean column identifying whether or not the integer is greater than 13. `return_tbl()` utilizes GPPC API reporting and SRF functions and macros.

You will develop the GPPC function on your Greenplum Database master host. Deploying the GPPC shared library that you create in this example requires administrative access to your Greenplum Database cluster.

Perform the following procedure to run the example:

1. Log in to the Greenplum Database master host and set up your environment. For example:

```
$ ssh gpadmin@<gpmaster>
gpadmin@gpmaster$ . /usr/local/greenplum-db/greenplum_path.sh
```

2. Create a work directory and navigate to the new directory. For example:

```
gpadmin@gpmaster$ mkdir gppc_work
gpadmin@gpmaster$ cd gppc_work
```

3. Prepare a source file for GPPC code by opening the file in the editor of your choice. For example, to open a file named `gppc_concat.c` using `vi`:

```
gpadmin@gpmaster$ vi gppc_rettbl.c
```

4. Copy/paste the following code into the file:

```
#include <stdio.h>
#include <string.h>
#include "gppc.h"

// initialize the logging level
GppcReportLevel level = GPPC_INFO;

// make the function SQL-invokable and declare the function
GPPC_FUNCTION_INFO(return_tbl);
GppcDatum return_tbl(GPPC_FUNCTION_ARGS);

GppcDatum
return_tbl(GPPC_FUNCTION_ARGS)
{
    GppcFuncCallContext fctx;
```

```
    GppcAnyTable intbl;
    GppcHeapTuple intuple;
    GppcTupleDesc in_tupdesc, out_tupdesc;
    GppcBool    resbool = false;
    GppcDatum    result, boolres, values[2];
    bool  nulls[2] = {false, false};

    // single input argument - the table
    intbl = GPPC_GETARG_ANYTABLE(0);

    // set the function context
    if (GPPC_SRF_IS_FIRSTCALL()) {
        fctx = GPPC_SRF_FIRSTCALL_INIT();
    }
    fctx = GPPC_SRF_PERCALL_SETUP();

    // get the tuple descriptor for the input table
    in_tupdesc  = GppcAnyTableGetTupleDesc(intbl);

    // retrieve the next tuple
    intuple = GppcAnyTableGetNextTuple(intbl);
    if( intuple == NULL ) {
      // no more tuples, conclude
      GPPC_SRF_RETURN_DONE(fctx);
    }

    // get the output tuple descriptor and verify that it is
    // defined as we expect
    out_tupdesc = GPPC_SRF_RESULT_DESC();
    if (GppcTupleDescNattrs(out_tupdesc) != 2                ||
        GppcTupleDescAttrType(out_tupdesc, 0) != GppcOidInt4 ||
        GppcTupleDescAttrType(out_tupdesc, 1) != GppcOidBool) {
        GppcReport(GPPC_ERROR, "INVALID out_tupdesc tuple");
    }

    // log the attribute names of the output tuple descriptor
    GppcReport(level, "output tuple descriptor attr0 name: %s",
GppcTupleDescAttrName(out_tupdesc, 0));
    GppcReport(level, "output tuple descriptor attr1 name: %s",
GppcTupleDescAttrName(out_tupdesc, 1));

    // retrieve the attribute values by name from the tuple
    bool text_isnull, int_isnull;
    GppcDatum intdat = GppcGetAttributeByName(intuple, "id", &int_isnull);
    GppcDatum textdat = GppcGetAttributeByName(intuple, "msg",
&text_isnull);

    // convert datum to specific type
    GppcInt4 intarg = GppcDatumGetInt4(intdat);
    GppcReport(level, "id: %d", intarg);
    GppcReport(level, "msg: %s",
GppcTextGetCString(GppcDatumGetText(textdat)));

    // perform the >13 check on the integer
    if( !int_isnull && (intarg > 13) ) {
        // greater than 13?
        resbool = true;
        GppcReport(level, "id is greater than 13!");
    }

    // values are datums; use integer from the tuple and
    // construct the datum for the boolean return
    values[0] = intdat;
    boolres = GppcBoolGetDatum(resbool);
```

```
    values[1] = boolres;

    // build a datum tuple and return
    result = GppcBuildHeapTupleDatum(out_tupdesc, values, nulls);
    GPPC_SRF_RETURN_NEXT(fctx, result);

}
```

The code declares and implements the `return_tbl()` function. It uses GPPC data types, macros, and functions to fetch the function arguments, examine tuple descriptors, build the return tuple, and return the result. The function also uses the SRF macros to keep track of the tuple context across function calls.

5. Save the file and exit the editor.
6. Open a file named `Makefile` in the editor of your choice. Copy/paste the following text into the file:

```
MODULE_big = gppc_rettbl
OBJS = gppc_rettbl.o

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)

PG_CPPFLAGS = -I$(shell $(PG_CONFIG) --includedir)
SHLIB_LINK = -L$(shell $(PG_CONFIG) --libdir) -lgppc
include $(PGXS)
```

7. Save the file and exit the editor.
8. Build a GPPC shared library for the `return_tbl()` function. For example:

```
gpadmin@gpmaster$ make all
```

The `make` command generates a shared library file named `gppc_rettbl.so` in the current working directory.

9. Copy the shared library to your Greenplum Database installation. You must have Greenplum Database administrative privileges to copy the file. For example:

```
gpadmin@gpmaster$ cp gppc_rettbl.so /usr/local/greenplum-db/lib/
postgresql/
```

This command copies the shared library to `$libdir`

10. Copy the shared library to every host in your Greenplum Database installation. For example, if `seghostfile` contains a list, one-host-per-line, of the segment hosts in your Greenplum Database cluster:

```
gpadmin@gpmaster$ gpscp -v -f seghostfile /usr/local/greenplum-db/lib/
postgresql/gppc_rettbl.so =:/usr/local/greenplum-db/lib/postgresql/
gppc_rettbl.so
```

11. Create a `.sql` file to register the GPPC `return_tbl()` function. Open a file named `gppc_rettbl_reg.sql` in the editor of your choice.
12. Copy/paste the following text into the file:

```
CREATE FUNCTION rettbl_gppc(anytable) RETURNS TABLE(id int4, thirteen
 bool)
  AS 'gppc_rettbl', 'return_tbl'
LANGUAGE C STRICT;
```

13. Register the GPPC function by running the script you just created. For example, to register the function in a database named `testdb`:

```
gpadmin@gpmaster$ psql -d testdb -f gppc_rettbl_reg.sql
```

14. Open a `psql` session. For example:

```
gpadmin@gpmaster$ psql -d testdb
```

15. Create a table with some test data. For example:

```
CREATE TABLE gppc_testtbl( id int, msg text );
INSERT INTO gppc_testtbl VALUES (1, 'f1');
INSERT INTO gppc_testtbl VALUES (7, 'f7');
INSERT INTO gppc_testtbl VALUES (10, 'f10');
INSERT INTO gppc_testtbl VALUES (13, 'f13');
INSERT INTO gppc_testtbl VALUES (15, 'f15');
INSERT INTO gppc_testtbl VALUES (17, 'f17');
```

16. Run the `rettbl_gppc()` function. For example:

```
testdb=# SELECT * FROM rettbl_gppc(TABLE(SELECT * FROM gppc_testtbl));
 id | thirteen
----+----------
  1 | f
  7 | f
 13 | f
 15 | t
 17 | t
 10 | f
(6 rows)
```

Developing a Background Worker Process

Greenplum Database can be extended to run user-supplied code in separate processes. Such processes are started, stopped, and monitored by `postgres`, which permits them to have a lifetime closely linked to the server's status. These processes have the option to attach to Greenplum Database's shared memory area and to connect to databases internally; they can also run multiple transactions serially, just like a regular client-connected server process. Also, by linking to `libpq` they can connect to the server and behave like a regular client application.

> **Warning:** There are considerable robustness and security risks in using background worker processes because, being written in the `C` language, they have unrestricted access to data. Administrators wishing to enable modules that include background worker processes should exercise extreme caution. Only carefully audited modules should be permitted to run background worker processes.

Background workers can be initialized at the time that Greenplum Database is started by including the module name in the *shared_preload_libraries* server configuration parameter. A module wishing to run a background worker can register it by calling `RegisterBackgroundWorker(BackgroundWorker *worker)` from its `_PG_init()`. Background workers can also be started after the system is up and running by calling the function `RegisterDynamicBackgroundWorker(BackgroundWorker *worker, BackgroundWorkerHandle **handle)`. Unlike `RegisterBackgroundWorker`, which can only be called from within the `postmaster`, `RegisterDynamicBackgroundWorker` must be called from a regular backend.

The structure `BackgroundWorker` is defined thus:

```
typedef void (*bgworker_main_type)(Datum main_arg);
typedef struct BackgroundWorker
```

```
{
    char        bgw_name[BGW_MAXLEN];
    int         bgw_flags;
    BgWorkerStartTime bgw_start_time;
    int         bgw_restart_time;       /* in seconds, or BGW_NEVER_RESTART
 */
    bgworker_main_type bgw_main;
    char        bgw_library_name[BGW_MAXLEN];   /* only if bgw_main is NULL
 */
    char        bgw_function_name[BGW_MAXLEN];  /* only if bgw_main is NULL
 */
    Datum       bgw_main_arg;
    int         bgw_notify_pid;
} BackgroundWorker;
```

`bgw_name` is a string to be used in log messages, process listings and similar contexts.

`bgw_flags` is a bitwise-or'd bit mask indicating the capabilities that the module wants. Possible values are `BGWORKER_SHMEM_ACCESS` (requesting shared memory access) and `BGWORKER_BACKEND_DATABASE_CONNECTION` (requesting the ability to establish a database connection, through which it can later run transactions and queries). A background worker using `BGWORKER_BACKEND_DATABASE_CONNECTION` to connect to a database must also attach shared memory using `BGWORKER_SHMEM_ACCESS`, or worker start-up will fail.

`bgw_start_time` is the server state during which `postgres` should start the process; it can be one of `BgWorkerStart_PostmasterStart` (start as soon as `postgres` itself has finished its own initialization; processes requesting this are not eligible for database connections), `BgWorkerStart_ConsistentState` (start as soon as a consistent state has been reached in a hot standby, allowing processes to connect to databases and run read-only queries), and `BgWorkerStart_RecoveryFinished` (start as soon as the system has entered normal read-write state). Note the last two values are equivalent in a server that's not a hot standby. Note that this setting only indicates when the processes are to be started; they do not stop when a different state is reached.

`bgw_restart_time` is the interval, in seconds, that `postgres` should wait before restarting the process, in case it crashes. It can be any positive value, or `BGW_NEVER_RESTART`, indicating not to restart the process in case of a crash.

`bgw_main` is a pointer to the function to run when the process is started. This function must take a single argument of type `Datum` and return `void`. `bgw_main_arg` will be passed to it as its only argument. Note that the global variable `MyBgworkerEntry` points to a copy of the `BackgroundWorker` structure passed at registration time. `bgw_main` may be NULL; in that case, `bgw_library_name` and `bgw_function_name` will be used to determine the entry point. This is useful for background workers launched after postmaster startup, where the postmaster does not have the requisite library loaded.

`bgw_library_name` is the name of a library in which the initial entry point for the background worker should be sought. It is ignored unless `bgw_main` is NULL. But if `bgw_main` is NULL, then the named library will be dynamically loaded by the worker process and `bgw_function_name` will be used to identify the function to be called.

`bgw_function_name` is the name of a function in a dynamically loaded library which should be used as the initial entry point for a new background worker. It is ignored unless `bgw_main` is NULL.

`bgw_notify_pid` is the PID of a Greenplum Database backend process to which the postmaster should send `SIGUSR1` when the process is started or exits. It should be 0 for workers registered at postmaster startup time, or when the backend registering the worker does not wish to wait for the worker to start up. Otherwise, it should be initialized to `MyProcPid`.

Once running, the process can connect to a database by calling `BackgroundWorkerInitializeConnection(char *dbname, char *username)`. This allows the process to run transactions and queries using the `SPI` interface. If *dbname* is NULL, the session is not connected to any particular database, but shared catalogs can be accessed. If *username* is NULL, the

process will run as the superuser created during `initdb`. BackgroundWorkerInitializeConnection can only be called once per background process, it is not possible to switch databases.

Signals are initially blocked when control reaches the `bgw_main` function, and must be unblocked by it; this is to allow the process to customize its signal handlers, if necessary. Signals can be unblocked in the new process by calling `BackgroundWorkerUnblockSignals` and blocked by calling `BackgroundWorkerBlockSignals`.

If `bgw_restart_time` for a background worker is configured as `BGW_NEVER_RESTART`, or if it exits with an exit code of 0 or is terminated by `TerminateBackgroundWorker`, it will be automatically unregistered by the postmaster on exit. Otherwise, it will be restarted after the time period configured via `bgw_restart_time`, or immediately if the postmaster reinitializes the cluster due to a backend failure. Backends which need to suspend execution only temporarily should use an interruptible sleep rather than exiting; this can be achieved by calling `WaitLatch()`. Make sure the `WL_POSTMASTER_DEATH` flag is set when calling that function, and verify the return code for a prompt exit in the emergency case that `postgres` itself has terminated.

When a background worker is registered using the `RegisterDynamicBackgroundWorker` function, it is possible for the backend performing the registration to obtain information regarding the status of the worker. Backends wishing to do this should pass the address of a `BackgroundWorkerHandle *` as the second argument to `RegisterDynamicBackgroundWorker`. If the worker is successfully registered, this pointer will be initialized with an opaque handle that can subsequently be passed to `GetBackgroundWorkerPid(BackgroundWorkerHandle *, pid_t *)` or `TerminateBackgroundWorker(BackgroundWorkerHandle *)`. `GetBackgroundWorkerPid` can be used to poll the status of the worker: a return value of `BGWH_NOT_YET_STARTED` indicates that the worker has not yet been started by the postmaster; `BGWH_STOPPED` indicates that it has been started but is no longer running; and `BGWH_STARTED` indicates that it is currently running. In this last case, the PID will also be returned via the second argument. `TerminateBackgroundWorker` causes the postmaster to send `SIGTERM` to the worker if it is running, and to unregister it as soon as it is not.

In some cases, a process which registers a background worker may wish to wait for the worker to start up. This can be accomplished by initializing `bgw_notify_pid` to `MyProcPid` and then passing the `BackgroundWorkerHandle *` obtained at registration time to `WaitForBackgroundWorkerStartup(BackgroundWorkerHandle *handle, pid_t *)` function. This function will block until the postmaster has attempted to start the background worker, or until the postmaster dies. If the background runner is running, the return value will `BGWH_STARTED`, and the PID will be written to the provided address. Otherwise, the return value will be `BGWH_STOPPED` or `BGWH_POSTMASTER_DIED`.

The `worker_spi` contrib module contains a working example, which demonstrates some useful techniques.

The maximum number of registered background workers is limited by max-worker-processes.

SQL Features, Reserved and Key Words, and Compliance

This section includes topics that identify SQL features and compliance in Greenplum Database:

Summary of Greenplum Features

This section provides a high-level overview of the system requirements and feature set of Greenplum Database. It contains the following topics:

Greenplum SQL Standard Conformance

The SQL language was first formally standardized in 1986 by the American National Standards Institute (ANSI) as SQL 1986. Subsequent versions of the SQL standard have been released by ANSI and as International Organization for Standardization (ISO) standards: SQL 1989, SQL 1992, SQL 1999, SQL 2003, SQL 2006, and finally SQL 2008, which is the current SQL standard. The official name of the standard is ISO/IEC 9075-14:2008. In general, each new version adds more features, although occasionally features are deprecated or removed.

It is important to note that there are no commercial database systems that are fully compliant with the SQL standard. Greenplum Database is almost fully compliant with the SQL 1992 standard, with most of the features from SQL 1999. Several features from SQL 2003 have also been implemented (most notably the SQL OLAP features).

This section addresses the important conformance issues of Greenplum Database as they relate to the SQL standards. For a feature-by-feature list of Greenplum's support of the latest SQL standard, see *SQL 2008 Optional Feature Compliance*.

Core SQL Conformance

In the process of building a parallel, shared-nothing database system and query optimizer, certain common SQL constructs are not currently implemented in Greenplum Database. The following SQL constructs are not supported:

1. Some set returning subqueries in `EXISTS` or `NOT EXISTS` clauses that Greenplum's parallel optimizer cannot rewrite into joins.
2. Backwards scrolling cursors, including the use of `FETCH PRIOR`, `FETCH FIRST`, `FETCH ABSOLUTE`, and `FETCH RELATIVE`.
3. In `CREATE TABLE` statements (on hash-distributed tables): a `UNIQUE` or `PRIMARY KEY` clause must include all of (or a superset of) the distribution key columns. Because of this restriction, only one `UNIQUE` clause or `PRIMARY KEY` clause is allowed in a `CREATE TABLE` statement. `UNIQUE` or `PRIMARY KEY` clauses are not allowed on randomly-distributed tables.
4. `CREATE UNIQUE INDEX` statements that do not contain all of (or a superset of) the distribution key columns. `CREATE UNIQUE INDEX` is not allowed on randomly-distributed tables.

   Note that `UNIQUE INDEXES` (but not `UNIQUE CONSTRAINTS`) are enforced on a part basis within a partitioned table. They guarantee the uniqueness of the key within each part or sub-part.
5. `VOLATILE` or `STABLE` functions cannot execute on the segments, and so are generally limited to being passed literal values as the arguments to their parameters.
6. Triggers are not supported since they typically rely on the use of `VOLATILE` functions.

7. Referential integrity constraints (foreign keys) are not enforced in Greenplum Database. Users can declare foreign keys and this information is kept in the system catalog, however.
8. Sequence manipulation functions CURRVAL and LASTVAL.

SQL 1992 Conformance

The following features of SQL 1992 are not supported in Greenplum Database:

1. NATIONAL CHARACTER (NCHAR) and NATIONAL CHARACTER VARYING (NVARCHAR). Users can declare the NCHAR and NVARCHAR types, however they are just synonyms for CHAR and VARCHAR in Greenplum Database.
2. CREATE ASSERTION statement.
3. INTERVAL literals are supported in Greenplum Database, but do not conform to the standard.
4. GET DIAGNOSTICS statement.
5. GLOBAL TEMPORARY TABLEs and LOCAL TEMPORARY TABLEs. Greenplum TEMPORARY TABLEs do not conform to the SQL standard, but many commercial database systems have implemented temporary tables in the same way. Greenplum temporary tables are the same as VOLATILE TABLEs in Teradata.
6. UNIQUE predicate.
7. MATCH PARTIAL for referential integrity checks (most likely will not be implemented in Greenplum Database).

SQL 1999 Conformance

The following features of SQL 1999 are not supported in Greenplum Database:

1. Large Object data types: BLOB, CLOB, NCLOB. However, the BYTEA and TEXT columns can store very large amounts of data in Greenplum Database (hundreds of megabytes).
2. MODULE (SQL client modules).
3. CREATE PROCEDURE (SQL/PSM). This can be worked around in Greenplum Database by creating a FUNCTION that returns void, and invoking the function as follows:

   ```
   SELECT myfunc(args);
   ```

4. The PostgreSQL/Greenplum function definition language (PL/PGSQL) is a subset of Oracle's PL/SQL, rather than being compatible with the SQL/PSM function definition language. Greenplum Database also supports function definitions written in Python, Perl, Java, and R.
5. BIT and BIT VARYING data types (intentionally omitted). These were deprecated in SQL 2003, and replaced in SQL 2008.
6. Greenplum supports identifiers up to 63 characters long. The SQL standard requires support for identifiers up to 128 characters long.
7. Prepared transactions (PREPARE TRANSACTION, COMMIT PREPARED, ROLLBACK PREPARED). This also means Greenplum does not support XA Transactions (2 phase commit coordination of database transactions with external transactions).
8. CHARACTER SET option on the definition of CHAR( ) or VARCHAR( ) columns.
9. Specification of CHARACTERS or OCTETS (BYTES) on the length of a CHAR( ) or VARCHAR( ) column. For example, VARCHAR(15 CHARACTERS) or VARCHAR(15 OCTETS) or VARCHAR(15 BYTES).
10. CURRENT_SCHEMA function.
11. CREATE DISTINCT TYPE statement. CREATE DOMAIN can be used as a work-around in Greenplum.
12. The *explicit table* construct.

SQL 2003 Conformance

The following features of SQL 2003 are not supported in Greenplum Database:

1. MERGE statements.

2. `IDENTITY` columns and the associated `GENERATED ALWAYS/GENERATED BY DEFAULT` clause. The `SERIAL` or `BIGSERIAL` data types are very similar to `INT` or `BIGINT GENERATED BY DEFAULT AS IDENTITY`.
3. `MULTISET` modifiers on data types.
4. `ROW` data type.
5. Greenplum Database syntax for using sequences is non-standard. For example, `nextval('seq')` is used in Greenplum instead of the standard `NEXT VALUE FOR seq`.
6. `GENERATED ALWAYS AS` columns. Views can be used as a work-around.
7. The sample clause (`TABLESAMPLE`) on `SELECT` statements. The `random()` function can be used as a work-around to get random samples from tables.
8. The *partitioned join tables* construct (`PARTITION BY` in a join).
9. For `CREATE TABLE x (LIKE(y))` statements, Greenplum does not support the `[INCLUDING|EXCLUDING][DEFAULTS|CONSTRAINTS|INDEXES]` clauses.
10. Greenplum array data types are almost SQL standard compliant with some exceptions. Generally customers should not encounter any problems using them.

SQL 2008 Conformance

The following features of SQL 2008 are not supported in Greenplum Database:

1. `BINARY` and `VARBINARY` data types. `BYTEA` can be used in place of `VARBINARY` in Greenplum Database.
2. `FETCH FIRST` or `FETCH NEXT` clause for `SELECT`, for example:

   ```
   SELECT id, name FROM tab1 ORDER BY id OFFSET 20 ROWS FETCH
   NEXT 10 ROWS ONLY;
   ```

   Greenplum has `LIMIT` and `LIMIT OFFSET` clauses instead.
3. The `ORDER BY` clause is ignored in views and subqueries unless a `LIMIT` clause is also used. This is intentional, as the Greenplum optimizer cannot determine when it is safe to avoid the sort, causing an unexpected performance impact for such `ORDER BY` clauses. To work around, you can specify a really large `LIMIT`. For example: `SELECT * FROM mytable ORDER BY 1 LIMIT 9999999999`
4. The *row subquery* construct is not supported.
5. `TRUNCATE TABLE` does not accept the `CONTINUE IDENTITY` and `RESTART IDENTITY` clauses.

Greenplum and PostgreSQL Compatibility

Greenplum Database is based on PostgreSQL 9.4. To support the distributed nature and typical workload of a Greenplum Database system, some SQL commands have been added or modified, and there are a few PostgreSQL features that are not supported. Greenplum has also added features not found in PostgreSQL, such as physical data distribution, parallel query optimization, external tables, resource queues, and enhanced table partitioning. For full SQL syntax and references, see the *SQL Commands*.

> **Note:** Greenplum Database does not support the PostgreSQL *large object facility* for streaming user data that is stored in large-object structures.

> **Note:** Pivotal does not support using `WITH OIDS` or `oids=TRUE` to assign an OID system column when creating or altering a table. This syntax is deprecated and will be removed in a future Greenplum release.

Table 260: SQL Support in Greenplum Database

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
`ALTER AGGREGATE`	YES	

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
ALTER CONVERSION	YES	
ALTER DATABASE	YES	
ALTER DOMAIN	YES	
ALTER EXTENSION	YES	Changes the definition of a Greenplum Database extension - based on PostgreSQL 9.6.
ALTER FUNCTION	YES	
ALTER GROUP	YES	An alias for *ALTER ROLE*
ALTER INDEX	YES	
ALTER LANGUAGE	YES	
ALTER OPERATOR	YES	
ALTER OPERATOR CLASS	YES	
ALTER OPERATOR FAMILY	YES	
ALTER PROTOCOL	YES	
ALTER RESOURCE QUEUE	YES	Greenplum Database resource management feature - not in PostgreSQL.
ALTER ROLE	YES	**Greenplum Database Clauses:** RESOURCE QUEUE *queue_name* | none
ALTER SCHEMA	YES	
ALTER SEQUENCE	YES	
ALTER SYSTEM	**NO**	
ALTER TABLE	YES	**Unsupported Clauses / Options:** CLUSTER ON ENABLE/DISABLE TRIGGER **Greenplum Database Clauses:** ADD | DROP | RENAME | SPLIT | EXCHANGE PARTITION | SET SUBPARTITION TEMPLATE | SET WITH (REORGANIZE= true | false) | SET DISTRIBUTED BY
ALTER TABLESPACE	YES	
ALTER TRIGGER	**NO**	

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
ALTER TYPE	YES	**Greenplum Database Clauses:** SET DEFAULT ENCODING
ALTER USER	YES	An alias for *ALTER ROLE*
ALTER VIEW	YES	
ANALYZE	YES	
BEGIN	YES	
CHECKPOINT	YES	
CLOSE	YES	
CLUSTER	YES	
COMMENT	YES	
COMMIT	YES	
COMMIT PREPARED	**NO**	
COPY	YES	**Modified Clauses:** ESCAPE [AS] '*escape*' | 'OFF' **Greenplum Database Clauses:** [LOG ERRORS] SEGMENT REJECT LIMIT *count* [ROWS| PERCENT]
CREATE AGGREGATE	YES	**Unsupported Clauses / Options:** [, SORTOP = *sort_operator*] **Greenplum Database Clauses:** [, COMBINEFUNC = *combinefunc*] **Limitations:** The functions used to implement the aggregate must be IMMUTABLE functions.
CREATE CAST	YES	
CREATE CONSTRAINT TRIGGER	**NO**	
CREATE CONVERSION	YES	
CREATE DATABASE	YES	
CREATE DOMAIN	YES	
CREATE EXTENSION	YES	Loads a new extension into Greenplum Database - based on PostgreSQL 9.6.

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
CREATE EXTERNAL TABLE	YES	Greenplum Database parallel ETL feature - not in PostgreSQL 9.4.
CREATE FUNCTION	YES	**Limitations:** Functions defined as STABLE or VOLATILE can be executed in Greenplum Database provided that they are executed on the master only. STABLE and VOLATILE functions cannot be used in statements that execute at the segment level.
CREATE GROUP	YES	An alias for *CREATE ROLE*
CREATE INDEX	YES	**Greenplum Database Clauses:** USING bitmap (bitmap indexes) **Limitations:** UNIQUE indexes are allowed only if they contain all of (or a superset of) the Greenplum distribution key columns. On partitioned tables, a unique index is only supported within an individual partition - not across all partitions. CONCURRENTLY keyword not supported in Greenplum.
CREATE LANGUAGE	YES	
CREATE MATERIALIZED VIEW	YES	Based on PostgreSQL 9.4.
CREATE OPERATOR	YES	**Limitations:** The function used to implement the operator must be an IMMUTABLE function.
CREATE OPERATOR CLASS	YES	
CREATE OPERATOR FAMILY	YES	
CREATE PROTOCOL	YES	
CREATE RESOURCE QUEUE	YES	Greenplum Database resource management feature - not in PostgreSQL 9.4.
CREATE ROLE	YES	**Greenplum Database Clauses:** RESOURCE QUEUE *queue_name* | none
CREATE RULE	YES	

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
CREATE SCHEMA	YES	
CREATE SEQUENCE	YES	**Limitations:** The `lastval()` and `currval()` functions are not supported. The `setval()` function is only allowed in queries that do not operate on distributed data.

CREATE TABLE	YES	**Unsupported Clauses / Options:**
		`[GLOBAL | LOCAL]`
		`REFERENCES`
		`FOREIGN KEY`
		`[DEFERRABLE | NOT DEFERRABLE]`
		Limited Clauses:
		`UNIQUE` or `PRIMARY KEY` constraints are only allowed on hash-distributed tables (`DISTRIBUTED BY`), and the constraint columns must be the same as or a superset of the distribution key columns of the table and must include all the distribution key columns of the partitioning key.
		Greenplum Database Clauses:
		`DISTRIBUTED BY (`*`column`*`, [ ... ] ) |`
		`DISTRIBUTED RANDOMLY`
		`PARTITION BY `*`type`*` (`*`column`*` [, ...])  ( `*`partition_specification`*`, [...] )`
		`WITH (appendoptimized=true [,compresslevel=value,blocksize=value] )`

CREATE TABLE AS	YES	See *CREATE TABLE*
CREATE TABLESPACE	**YES**	**Greenplum Database Clauses:** Specify host file system locations for specific segment instances. `WITH (content`*`ID_1`*`='`*`/path/to/dir1`*`...)`

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
CREATE TRIGGER	**NO**	
CREATE TYPE	YES	**Greenplum Database Clauses:** COMPRESSTYPE | COMPRESSLEVEL | BLOCKSIZE **Limitations:** The functions used to implement a new base type must be IMMUTABLE functions.
CREATE USER	YES	An alias for *CREATE ROLE*
CREATE VIEW	YES	
DEALLOCATE	YES	
DECLARE	YES	**Unsupported Clauses / Options:** SCROLL FOR UPDATE [OF column [, ...]] **Limitations:** Cursors cannot be backward-scrolled. Forward scrolling is supported. PL/pgSQL does not have support for updatable cursors.
DELETE	YES	
DISCARD	YES	**Limitation:** DISCARD ALL is not supported.
DO	YES	PostgreSQL 9.0 feature
DROP AGGREGATE	YES	
DROP CAST	YES	
DROP CONVERSION	YES	
DROP DATABASE	YES	
DROP DOMAIN	YES	
DROP EXTENSION	YES	Removes an extension from Greenplum Database – based on PostgreSQL 9.6.
DROP EXTERNAL TABLE	YES	Greenplum Database parallel ETL feature - not in PostgreSQL 9.4.
DROP FUNCTION	YES	

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
DROP GROUP	YES	An alias for *DROP ROLE*
DROP INDEX	YES	
DROP LANGUAGE	YES	
DROP OPERATOR	YES	
DROP OPERATOR CLASS	YES	
DROP OPERATOR FAMILY	YES	
DROP OWNED	**NO**	
DROP PROTOCOL	YES	
DROP RESOURCE QUEUE	YES	Greenplum Database resource management feature - not in PostgreSQL 9.4.
DROP ROLE	YES	
DROP RULE	YES	
DROP SCHEMA	YES	
DROP SEQUENCE	YES	
DROP TABLE	YES	
DROP TABLESPACE	YES	
DROP TRIGGER	**NO**	
DROP TYPE	YES	
DROP USER	YES	An alias for *DROP ROLE*
DROP VIEW	YES	
END	YES	
EXECUTE	YES	
EXPLAIN	YES	
FETCH	YES	**Unsupported Clauses / Options:** LAST PRIOR BACKWARD BACKWARD ALL **Limitations:** Cannot fetch rows in a nonsequential fashion; backward scan is not supported.
GRANT	YES	
INSERT	YES	

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
LATERAL Join Type	**NO**	
LISTEN	**NO**	
LOAD	YES	
LOCK	YES	
MOVE	YES	See *FETCH*
NOTIFY	**NO**	
PREPARE	YES	
PREPARE TRANSACTION	**NO**	
REASSIGN OWNED	YES	
REFRESH MATERIALIZED VIEW	YES	Based on PostgreSQL 9.4.
REINDEX	YES	
RELEASE SAVEPOINT	YES	
RESET	YES	
REVOKE	YES	
ROLLBACK	YES	
ROLLBACK PREPARED	**NO**	
ROLLBACK TO SAVEPOINT	YES	
SAVEPOINT	YES	
SELECT	YES	**Limitations:** Limited use of VOLATILE and STABLE functions in FROM or WHERE clauses Text search (Tsearch2) is not supported FETCH FIRST or FETCH NEXT clauses not supported **Greenplum Database Clauses (OLAP):** [GROUP BY *grouping_element* [, ...]] [WINDOW *window_name* AS (*window_specification*)] [FILTER (WHERE *condition*)] applied to an aggregate function in the SELECT list
SELECT INTO	YES	See *SELECT*
SET	YES	

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
SET CONSTRAINTS	**NO**	In PostgreSQL, this only applies to foreign key constraints, which are currently not enforced in Greenplum Database.
SET ROLE	YES	
SET SESSION AUTHORIZATION	YES	Deprecated as of PostgreSQL 8.1 - see *SET ROLE*
SET TRANSACTION	YES	**Limitations:** DEFERRABLE clause has no effect. SET TRANSACTION SNAPSHOT command is not supported.
SHOW	YES	
START TRANSACTION	YES	
TRUNCATE	YES	
UNLISTEN	**NO**	
UPDATE	YES	**Limitations:** SET not allowed for Greenplum distribution key columns.
VACUUM	YES	**Limitations:** VACUUM FULL is not recommended in Greenplum Database.
VALUES	YES	

Reserved Identifiers and SQL Key Words

This topic describes Greenplum Database reserved identifiers and object names, and SQL key words recognized by the Greenplum Database and PostgreSQL command parsers.

Reserved Identifiers

In the Greenplum Database system, names beginning with `gp_` and `pg_` are reserved and should not be used as names for user-created objects, such as tables, views, and functions.

The resource group names `admin_group`, `default_group`, and `none` are reserved. The resource queue name `pg_default` is reserved.

The tablespace names `pg_default` and `pg_global` are reserved.

The role names `gpadmin` and `gpmon` are reserved. `gpadmin` is the default Greenplum Database superuser role. The `gpmon` role owns the `gpperfmon` database and is also used by Greenplum Command Center.

In data files, the characters that delimit fields (columns) and rows have a special meaning. If they appear within the data you must escape them so that Greenplum Database treats them as data and not as delimiters. The backslash character (\) is the default escape character. See *Escaping* for details.

See *SQL Syntax* in the PostgreSQL documentation for more information about SQL identifiers, constants, operators, and expressions.

SQL Key Words

Table 261: SQL Key Words lists all tokens that are key words in Greenplum Database 6 and PostgreSQL 9.4.

ANSI SQL distinguishes between *reserved* and *unreserved* key words. According to the standard, reserved key words are the only real key words; they are never allowed as identifiers. Unreserved key words only have a special meaning in particular contexts and can be used as identifiers in other contexts. Most unreserved key words are actually the names of built-in tables and functions specified by SQL. The concept of unreserved key words essentially only exists to declare that some predefined meaning is attached to a word in some contexts.

In the Greenplum Database and PostgreSQL parsers there are several different classes of tokens ranging from those that can never be used as an identifier to those that have absolutely no special status in the parser as compared to an ordinary identifier. (The latter is usually the case for functions specified by SQL.) Even reserved key words are not completely reserved, but can be used as column labels (for example, `SELECT 55 AS CHECK`, even though `CHECK` is a reserved key word).

Table 261: SQL Key Words classifies as "unreserved" those key words that are explicitly known to the parser but are allowed as column or table names. Some key words that are otherwise unreserved cannot be used as function or data type names and are marked accordingly. (Most of these words represent built-in functions or data types with special syntax. The function or type is still available but it cannot be redefined by the user.) Key words labeled "reserved" are not allowed as column or table names. Some reserved key words are allowable as names for functions or data types; this is also shown in the table. If not so marked, a reserved key word is only allowed as an "AS" column label name.

If you get spurious parser errors for commands that contain any of the listed key words as an identifier you should try to quote the identifier to see if the problem goes away.

Before studying the table, note the fact that a key word is not reserved does not mean that the feature related to the word is not implemented. Conversely, the presence of a key word does not indicate the existence of a feature.

Table 261: SQL Key Words

Key Word	Greenplum Database	PostgreSQL 9.4
ABORT	unreserved	unreserved
ABSOLUTE	unreserved	unreserved
ACCESS	unreserved	unreserved
ACTION	unreserved	unreserved
ACTIVE	unreserved	
ADD	unreserved	unreserved
ADMIN	unreserved	unreserved
AFTER	unreserved	unreserved
AGGREGATE	unreserved	unreserved
ALL	reserved	reserved

Key Word	Greenplum Database	PostgreSQL 9.4
ALSO	unreserved	unreserved
ALTER	unreserved	unreserved
ALWAYS	unreserved	unreserved
ANALYSE	reserved	reserved
ANALYZE	reserved	reserved
AND	reserved	reserved
ANY	reserved	reserved
ARRAY	reserved	reserved
AS	reserved	reserved
ASC	reserved	reserved
ASSERTION	unreserved	unreserved
ASSIGNMENT	unreserved	unreserved
ASYMMETRIC	reserved	reserved
AT	unreserved	unreserved
ATTRIBUTE	unreserved	unreserved
AUTHORIZATION	reserved (can be function or type name)	reserved (can be function or type name)
BACKWARD	unreserved	unreserved
BEFORE	unreserved	unreserved
BEGIN	unreserved	unreserved
BETWEEN	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
BIGINT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
BINARY	reserved (can be function or type name)	reserved (can be function or type name)
BIT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
BOOLEAN	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
BOTH	reserved	reserved
BY	unreserved	unreserved
CACHE	unreserved	unreserved
CALLED	unreserved	unreserved
CASCADE	unreserved	unreserved
CASCADED	unreserved	unreserved
CASE	reserved	reserved
CAST	reserved	reserved

Key Word	Greenplum Database	PostgreSQL 9.4
CATALOG	unreserved	unreserved
CHAIN	unreserved	unreserved
CHAR	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
CHARACTER	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
CHARACTERISTICS	unreserved	unreserved
CHECK	reserved	reserved
CHECKPOINT	unreserved	unreserved
CLASS	unreserved	unreserved
CLOSE	unreserved	unreserved
CLUSTER	unreserved	unreserved
COALESCE	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
COLLATE	reserved	reserved
COLLATION	reserved (can be function or type name)	reserved (can be function or type name)
COLUMN	reserved	reserved
COMMENT	unreserved	unreserved
COMMENTS	unreserved	unreserved
COMMIT	unreserved	unreserved
COMMITTED	unreserved	unreserved
CONCURRENCY	unreserved	
CONCURRENTLY	reserved (can be function or type name)	reserved (can be function or type name)
CONFIGURATION	unreserved	unreserved
CONNECTION	unreserved	unreserved
CONSTRAINT	reserved	reserved
CONSTRAINTS	unreserved	unreserved
CONTAINS	unreserved	
CONTENT	unreserved	unreserved
CONTINUE	unreserved	unreserved
CONVERSION	unreserved	unreserved
COPY	unreserved	unreserved
COST	unreserved	unreserved
CPU_RATE_LIMIT	unreserved	
CPUSET	unreserved	

Key Word	Greenplum Database	PostgreSQL 9.4
CREATE	reserved	reserved
CREATEEXTTABLE	unreserved	
CROSS	reserved (can be function or type name)	reserved (can be function or type name)
CSV	unreserved	unreserved
CUBE	unreserved (cannot be function or type name)	
CURRENT	unreserved	unreserved
CURRENT_CATALOG	reserved	reserved
CURRENT_DATE	reserved	reserved
CURRENT_ROLE	reserved	reserved
CURRENT_SCHEMA	reserved (can be function or type name)	reserved (can be function or type name)
CURRENT_TIME	reserved	reserved
CURRENT_TIMESTAMP	reserved	reserved
CURRENT_USER	reserved	reserved
CURSOR	unreserved	unreserved
CYCLE	unreserved	unreserved
DATA	unreserved	unreserved
DATABASE	unreserved	unreserved
DAY	unreserved	unreserved
DEALLOCATE	unreserved	unreserved
DEC	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
DECIMAL	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
DECLARE	unreserved	unreserved
DECODE	reserved	
DEFAULT	reserved	reserved
DEFAULTS	unreserved	unreserved
DEFERRABLE	reserved	reserved
DEFERRED	unreserved	unreserved
DEFINER	unreserved	unreserved
DELETE	unreserved	unreserved
DELIMITER	unreserved	unreserved

Key Word	Greenplum Database	PostgreSQL 9.4
DELIMITERS	unreserved	unreserved
DENY	unreserved	
DESC	reserved	reserved
DICTIONARY	unreserved	unreserved
DISABLE	unreserved	unreserved
DISCARD	unreserved	unreserved
DISTINCT	reserved	reserved
DISTRIBUTED	reserved	
DO	reserved	reserved
DOCUMENT	unreserved	unreserved
DOMAIN	unreserved	unreserved
DOUBLE	unreserved	unreserved
DROP	unreserved	unreserved
DXL	unreserved	
EACH	unreserved	unreserved
ELSE	reserved	reserved
ENABLE	unreserved	unreserved
ENCODING	unreserved	unreserved
ENCRYPTED	unreserved	unreserved
END	reserved	reserved
ENUM	unreserved	unreserved
ERRORS	unreserved	
ESCAPE	unreserved	unreserved
EVENT	unreserved	unreserved
EVERY	unreserved	
EXCEPT	reserved	reserved
EXCHANGE	unreserved	
EXCLUDE	reserved	unreserved
EXCLUDING	unreserved	unreserved
EXCLUSIVE	unreserved	unreserved
EXECUTE	unreserved	unreserved
EXISTS	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
EXPAND	unreserved	
EXPLAIN	unreserved	unreserved

Key Word	Greenplum Database	PostgreSQL 9.4
EXTENSION	unreserved	unreserved
EXTERNAL	unreserved	unreserved
EXTRACT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
FALSE	reserved	reserved
FAMILY	unreserved	unreserved
FETCH	reserved	reserved
FIELDS	unreserved	
FILL	unreserved	
FILTER	unreserved	unreserved
FIRST	unreserved	unreserved
FLOAT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
FOLLOWING	reserved	unreserved
FOR	reserved	reserved
FORCE	unreserved	unreserved
FOREIGN	reserved	reserved
FORMAT	unreserved	
FORWARD	unreserved	unreserved
FREEZE	reserved (can be function or type name)	reserved (can be function or type name)
FROM	reserved	reserved
FULL	reserved (can be function or type name)	reserved (can be function or type name)
FULLSCAN	unreserved	
FUNCTION	unreserved	unreserved
FUNCTIONS	unreserved	unreserved
GLOBAL	unreserved	unreserved
GRANT	reserved	reserved
GRANTED	unreserved	unreserved
GREATEST	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
GROUP	reserved	reserved
GROUP_ID	unreserved (cannot be function or type name)	
GROUPING	unreserved (cannot be function or type name)	
HANDLER	unreserved	unreserved

Key Word	Greenplum Database	PostgreSQL 9.4
HASH	unreserved	
HAVING	reserved	reserved
HEADER	unreserved	unreserved
HOLD	unreserved	unreserved
HOST	unreserved	
HOUR	unreserved	unreserved
IDENTITY	unreserved	unreserved
IF	unreserved	unreserved
IGNORE	unreserved	
ILIKE	reserved (can be function or type name)	reserved (can be function or type name)
IMMEDIATE	unreserved	unreserved
IMMUTABLE	unreserved	unreserved
IMPLICIT	unreserved	unreserved
IN	reserved	reserved
INCLUDING	unreserved	unreserved
INCLUSIVE	unreserved	
INCREMENT	unreserved	unreserved
INDEX	unreserved	unreserved
INDEXES	unreserved	unreserved
INHERIT	unreserved	unreserved
INHERITS	unreserved	unreserved
INITIALLY	reserved	reserved
INLINE	unreserved	unreserved
INNER	reserved (can be function or type name)	reserved (can be function or type name)
INOUT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
INPUT	unreserved	unreserved
INSENSITIVE	unreserved	unreserved
INSERT	unreserved	unreserved
INSTEAD	unreserved	unreserved
INT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
INTEGER	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
INTERSECT	reserved	reserved

Key Word	Greenplum Database	PostgreSQL 9.4
INTERVAL	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
INTO	reserved	reserved
INVOKER	unreserved	unreserved
IS	reserved (can be function or type name)	reserved (can be function or type name)
ISNULL	reserved (can be function or type name)	reserved (can be function or type name)
ISOLATION	unreserved	unreserved
JOIN	reserved (can be function or type name)	reserved (can be function or type name)
KEY	unreserved	unreserved
LABEL	unreserved	unreserved
LANGUAGE	unreserved	unreserved
LARGE	unreserved	unreserved
LAST	unreserved	unreserved
LATERAL	reserved	reserved
LC_COLLATE	unreserved	unreserved
LC_CTYPE	unreserved	unreserved
LEADING	reserved	reserved
LEAKPROOF	unreserved	unreserved
LEAST	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
LEFT	reserved (can be function or type name)	reserved (can be function or type name)
LEVEL	unreserved	unreserved
LIKE	reserved (can be function or type name)	reserved (can be function or type name)
LIMIT	reserved	reserved
LIST	unreserved	
LISTEN	unreserved	unreserved
LOAD	unreserved	unreserved
LOCAL	unreserved	unreserved
LOCALTIME	reserved	reserved
LOCALTIMESTAMP	reserved	reserved
LOCATION	unreserved	unreserved
LOCK	unreserved	unreserved
LOG	reserved (can be function or type name)	
MAPPING	unreserved	unreserved
MASTER	unreserved	

Key Word	Greenplum Database	PostgreSQL 9.4
MATCH	unreserved	unreserved
MATERIALIZED	unreserved	unreserved
MAXVALUE	unreserved	unreserved
MEDIAN	unreserved (cannot be function or type name)	
MEMORY_LIMIT	unreserved	
MEMORY_ SHARED_QUOTA	unreserved	
MEMORY_SPILL_ RATIO	unreserved	
MINUTE	unreserved	unreserved
MINVALUE	unreserved	unreserved
MISSING	unreserved	
MODE	unreserved	unreserved
MODIFIES	unreserved	
MONTH	unreserved	unreserved
MOVE	unreserved	unreserved
NAME	unreserved	unreserved
NAMES	unreserved	unreserved
NATIONAL	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
NATURAL	reserved (can be function or type name)	reserved (can be function or type name)
NCHAR	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
NEWLINE	unreserved	
NEXT	unreserved	unreserved
NO	unreserved	unreserved
NOCREATEEXTTABLE	unreserved	
NONE	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
NOOVERCOMMIT	unreserved	
NOT	reserved	reserved
NOTHING	unreserved	unreserved
NOTIFY	unreserved	unreserved
NOTNULL	reserved (can be function or type name)	reserved (can be function or type name)
NOWAIT	unreserved	unreserved

Key Word	Greenplum Database	PostgreSQL 9.4
NULL	reserved	reserved
NULLIF	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
NULLS	unreserved	unreserved
NUMERIC	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
OBJECT	unreserved	unreserved
OF	unreserved	unreserved
OFF	unreserved	unreserved
OFFSET	reserved	reserved
OIDS	unreserved	unreserved
ON	reserved	reserved
ONLY	reserved	reserved
OPERATOR	unreserved	unreserved
OPTION	unreserved	unreserved
OPTIONS	unreserved	unreserved
OR	reserved	reserved
ORDER	reserved	reserved
ORDERED	unreserved	
ORDINALITY	unreserved	unreserved
OTHERS	unreserved	
OUT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
OUTER	reserved (can be function or type name)	reserved (can be function or type name)
OVER	unreserved	unreserved
OVERCOMMIT	unreserved	
OVERLAPS	reserved (can be function or type name)	reserved (can be function or type name)
OVERLAY	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
OWNED	unreserved	unreserved
OWNER	unreserved	unreserved
PARSER	unreserved	unreserved
PARTIAL	unreserved	unreserved
PARTITION	reserved	unreserved
PARTITIONS	unreserved	
PASSING	unreserved	unreserved

Key Word	Greenplum Database	PostgreSQL 9.4
PASSWORD	unreserved	unreserved
PERCENT	unreserved	
PLACING	reserved	reserved
PLANS	unreserved	unreserved
POSITION	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
PRECEDING	reserved	unreserved
PRECISION	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
PREPARE	unreserved	unreserved
PREPARED	unreserved	unreserved
PRESERVE	unreserved	unreserved
PRIMARY	reserved	reserved
PRIOR	unreserved	unreserved
PRIVILEGES	unreserved	unreserved
PROCEDURAL	unreserved	unreserved
PROCEDURE	unreserved	unreserved
PROGRAM	unreserved	unreserved
PROTOCOL	unreserved	
QUEUE	unreserved	
QUOTE	unreserved	unreserved
RANDOMLY	unreserved	
RANGE	unreserved	unreserved
READ	unreserved	unreserved
READABLE	unreserved	
READS	unreserved	
REAL	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
REASSIGN	unreserved	unreserved
RECHECK	unreserved	unreserved
RECURSIVE	unreserved	unreserved
REF	unreserved	unreserved
REFERENCES	reserved	reserved
REFRESH	unreserved	unreserved
REINDEX	unreserved	unreserved

Key Word	Greenplum Database	PostgreSQL 9.4
REJECT	unreserved	
RELATIVE	unreserved	unreserved
RELEASE	unreserved	unreserved
RENAME	unreserved	unreserved
REPEATABLE	unreserved	unreserved
REPLACE	unreserved	unreserved
REPLICA	unreserved	unreserved
REPLICATED	unreserved	
RESET	unreserved	unreserved
RESOURCE	unreserved	
RESTART	unreserved	unreserved
RESTRICT	unreserved	unreserved
RETURNING	reserved	reserved
RETURNS	unreserved	unreserved
REVOKE	unreserved	unreserved
RIGHT	reserved (can be function or type name)	reserved (can be function or type name)
ROLE	unreserved	unreserved
ROLLBACK	unreserved	unreserved
ROLLUP	unreserved (cannot be function or type name)	
ROOTPARTITION	unreserved	
ROW	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
ROWS	unreserved	unreserved
RULE	unreserved	unreserved
SAVEPOINT	unreserved	unreserved
SCATTER	reserved	
SCHEMA	unreserved	unreserved
SCROLL	unreserved	unreserved
SEARCH	unreserved	unreserved
SECOND	unreserved	unreserved
SECURITY	unreserved	unreserved
SEGMENT	unreserved	
SEGMENTS	unreserved	
SELECT	reserved	reserved

Key Word	Greenplum Database	PostgreSQL 9.4
SEQUENCE	unreserved	unreserved
SEQUENCES	unreserved	unreserved
SERIALIZABLE	unreserved	unreserved
SERVER	unreserved	unreserved
SESSION	unreserved	unreserved
SESSION_USER	reserved	reserved
SET	unreserved	unreserved
SETOF	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
SETS	unreserved (cannot be function or type name)	
SHARE	unreserved	unreserved
SHOW	unreserved	unreserved
SIMILAR	reserved (can be function or type name)	reserved (can be function or type name)
SIMPLE	unreserved	unreserved
SMALLINT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
SNAPSHOT	unreserved	unreserved
SOME	reserved	reserved
SPLIT	unreserved	
SQL	unreserved	
STABLE	unreserved	unreserved
STANDALONE	unreserved	unreserved
START	unreserved	unreserved
STATEMENT	unreserved	unreserved
STATISTICS	unreserved	unreserved
STDIN	unreserved	unreserved
STDOUT	unreserved	unreserved
STORAGE	unreserved	unreserved
STRICT	unreserved	unreserved
STRIP	unreserved	unreserved
SUBPARTITION	unreserved	
SUBSTRING	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
SYMMETRIC	reserved	reserved
SYSID	unreserved	unreserved

Key Word	Greenplum Database	PostgreSQL 9.4
SYSTEM	unreserved	unreserved
TABLE	reserved	reserved
TABLES	unreserved	unreserved
TABLESPACE	unreserved	unreserved
TEMP	unreserved	unreserved
TEMPLATE	unreserved	unreserved
TEMPORARY	unreserved	unreserved
TEXT	unreserved	unreserved
THEN	reserved	reserved
THRESHOLD	unreserved	
TIES	unreserved	
TIME	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
TIMESTAMP	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
TO	reserved	reserved
TRAILING	reserved	reserved
TRANSACTION	unreserved	unreserved
TREAT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
TRIGGER	unreserved	unreserved
TRIM	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
TRUE	reserved	reserved
TRUNCATE	unreserved	unreserved
TRUSTED	unreserved	unreserved
TYPE	unreserved	unreserved
TYPES	unreserved	unreserved
UNBOUNDED	reserved	unreserved
UNCOMMITTED	unreserved	unreserved
UNENCRYPTED	unreserved	unreserved
UNION	reserved	reserved
UNIQUE	reserved	reserved
UNKNOWN	unreserved	unreserved
UNLISTEN	unreserved	unreserved
UNLOGGED	unreserved	unreserved

Key Word	Greenplum Database	PostgreSQL 9.4
UNTIL	unreserved	unreserved
UPDATE	unreserved	unreserved
USER	reserved	reserved
USING	reserved	reserved
VACUUM	unreserved	unreserved
VALID	unreserved	unreserved
VALIDATE	unreserved	unreserved
VALIDATION	unreserved	
VALIDATOR	unreserved	unreserved
VALUE	unreserved	unreserved
VALUES	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
VARCHAR	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
VARIADIC	reserved	reserved
VARYING	unreserved	unreserved
VERBOSE	reserved (can be function or type name)	reserved (can be function or type name)
VERSION	unreserved	unreserved
VIEW	unreserved	unreserved
VIEWS	unreserved	unreserved
VOLATILE	unreserved	unreserved
WEB	unreserved	
WHEN	reserved	reserved
WHERE	reserved	reserved
WHITESPACE	unreserved	unreserved
WINDOW	reserved	reserved
WITH	reserved	reserved
WITHIN	unreserved	unreserved
WITHOUT	unreserved	unreserved
WORK	unreserved	unreserved
WRAPPER	unreserved	unreserved
WRITABLE	unreserved	
WRITE	unreserved	unreserved
XML	unreserved	unreserved

Key Word	Greenplum Database	PostgreSQL 9.4
XMLATTRIBUTES	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
XMLCONCAT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
XMLELEMENT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
XMLEXISTS	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
XMLFOREST	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
XMLPARSE	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
XMLPI	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
XMLROOT	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
XMLSERIALIZE	unreserved (cannot be function or type name)	unreserved (cannot be function or type name)
YEAR	unreserved	unreserved
YES	unreserved	unreserved
ZONE	unreserved	unreserved

SQL 2008 Optional Feature Compliance

The following table lists the features described in the 2008 SQL standard. Features that are supported in Greenplum Database are marked as YES in the 'Supported' column, features that are not implemented are marked as NO.

For information about Greenplum features and SQL compliance, see the *Greenplum Database Administrator Guide*.

Table 262: SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
B011	Embedded Ada	NO	
B012	Embedded C	NO	Due to issues with PostgreSQL ecpg
B013	Embedded COBOL	NO	
B014	Embedded Fortran	NO	
B015	Embedded MUMPS	NO	
B016	Embedded Pascal	NO	
B017	Embedded PL/I	NO	
B021	Direct SQL	YES	

ID	Feature	Supported	Comments
B031	Basic dynamic SQL	NO	
B032	Extended dynamic SQL	NO	
B033	Untyped SQL-invoked function arguments	NO	
B034	Dynamic specification of cursor attributes	NO	
B035	Non-extended descriptor names	NO	
B041	Extensions to embedded SQL exception declarations	NO	
B051	Enhanced execution rights	NO	
B111	Module language Ada	NO	
B112	Module language C	NO	
B113	Module language COBOL	NO	
B114	Module language Fortran	NO	
B115	Module language MUMPS	NO	
B116	Module language Pascal	NO	
B117	Module language PL/I	NO	
B121	Routine language Ada	NO	
B122	Routine language C	NO	
B123	Routine language COBOL	NO	
B124	Routine language Fortran	NO	
B125	Routine language MUMPS	NO	
B126	Routine language Pascal	NO	
B127	Routine language PL/I	NO	
B128	Routine language SQL	NO	
E011	Numeric data types	YES	
E011-01	`INTEGER` and `SMALLINT` data types	YES	
E011-02	`DOUBLE PRECISION` and `FLOAT` data types	YES	

ID	Feature	Supported	Comments
E011-03	`DECIMAL` and `NUMERIC` data types	YES	
E011-04	Arithmetic operators	YES	
E011-05	Numeric comparison	YES	
E011-06	Implicit casting among the numeric data types	YES	
E021	Character data types	YES	
E021-01	`CHARACTER` data type	YES	
E021-02	`CHARACTER VARYING` data type	YES	
E021-03	Character literals	YES	
E021-04	`CHARACTER_LENGTH` function	YES	Trims trailing spaces from `CHARACTER` values before counting
E021-05	`OCTET_LENGTH` function	YES	
E021-06	`SUBSTRING` function	YES	
E021-07	Character concatenation	YES	
E021-08	`UPPER` and `LOWER` functions	YES	
E021-09	`TRIM` function	YES	
E021-10	Implicit casting among the character string types	YES	
E021-11	`POSITION` function	YES	
E021-12	Character comparison	YES	
E031	Identifiers	YES	
E031-01	Delimited identifiers	YES	
E031-02	Lower case identifiers	YES	
E031-03	Trailing underscore	YES	
E051	Basic query specification	YES	
E051-01	`SELECT DISTINCT`	YES	
E051-02	`GROUP BY` clause	YES	
E051-03	`GROUP BY` can contain columns not in `SELECT` list	YES	
E051-04	`SELECT` list items can be renamed	YES	
E051-05	`HAVING` clause	YES	

ID	Feature	Supported	Comments
E051-06	Qualified * in `SELECT` list	YES	
E051-07	Correlation names in the `FROM` clause	YES	
E051-08	Rename columns in the `FROM` clause	YES	
E061	Basic predicates and search conditions	YES	
E061-01	Comparison predicate	YES	
E061-02	`BETWEEN` predicate	YES	
E061-03	`IN` predicate with list of values	YES	
E061-04	`LIKE` predicate	YES	
E061-05	`LIKE` predicate `ESCAPE` clause	YES	
E061-06	`NULL` predicate	YES	
E061-07	Quantified comparison predicate	YES	
E061-08	`EXISTS` predicate	YES	Not all uses work in Greenplum
E061-09	Subqueries in comparison predicate	YES	
E061-11	Subqueries in IN predicate	YES	
E061-12	Subqueries in quantified comparison predicate	YES	
E061-13	Correlated subqueries	YES	
E061-14	Search condition	YES	
E071	Basic query expressions	YES	
E071-01	`UNION DISTINCT` table operator	YES	
E071-02	`UNION ALL` table operator	YES	
E071-03	`EXCEPT DISTINCT` table operator	YES	
E071-05	Columns combined via table operators need not have exactly the same data type	YES	
E071-06	Table operators in subqueries	YES	

ID	Feature	Supported	Comments
E081	Basic Privileges	NO	Partial sub-feature support
E081-01	`SELECT` privilege	YES	
E081-02	`DELETE` privilege	YES	
E081-03	`INSERT` privilege at the table level	YES	
E081-04	`UPDATE` privilege at the table level	YES	
E081-05	`UPDATE` privilege at the column level	YES	
E081-06	`REFERENCES` privilege at the table level	NO	
E081-07	`REFERENCES` privilege at the column level	NO	
E081-08	`WITH GRANT OPTION`	YES	
E081-09	`USAGE` privilege	YES	
E081-10	`EXECUTE` privilege	YES	
E091	Set Functions	YES	
E091-01	`AVG`	YES	
E091-02	`COUNT`	YES	
E091-03	`MAX`	YES	
E091-04	`MIN`	YES	
E091-05	`SUM`	YES	
E091-06	`ALL` quantifier	YES	
E091-07	`DISTINCT` quantifier	YES	
E101	Basic data manipulation	YES	
E101-01	`INSERT` statement	YES	
E101-03	Searched `UPDATE` statement	YES	
E101-04	Searched `DELETE` statement	YES	
E111	Single row `SELECT` statement	YES	
E121	Basic cursor support	YES	
E121-01	`DECLARE CURSOR`	YES	
E121-02	`ORDER BY` columns need not be in select list	YES	

ID	Feature	Supported	Comments
E121-03	Value expressions in `ORDER BY` clause	YES	
E121-04	`OPEN` statement	YES	
E121-06	Positioned `UPDATE` statement	NO	
E121-07	Positioned `DELETE` statement	NO	
E121-08	`CLOSE` statement	YES	
E121-10	`FETCH` statement implicit `NEXT`	YES	
E121-17	`WITH HOLD` cursors	YES	
E131	Null value support	YES	
E141	Basic integrity constraints	YES	
E141-01	`NOT NULL` constraints	YES	
E141-02	`UNIQUE` constraints of `NOT NULL` columns	YES	Must be the same as or a superset of the Greenplum distribution key
E141-03	`PRIMARY KEY` constraints	YES	Must be the same as or a superset of the Greenplum distribution key
E141-04	Basic `FOREIGN KEY` constraint with the `NO ACTION` default for both referential delete action and referential update action	NO	
E141-06	`CHECK` constraints	YES	
E141-07	Column defaults	YES	
E141-08	`NOT NULL` inferred on `PRIMARY KEY`	YES	
E141-10	Names in a foreign key can be specified in any order	YES	Foreign keys can be declared but are not enforced in Greenplum
E151	Transaction support	YES	
E151-01	`COMMIT` statement	YES	
E151-02	`ROLLBACK` statement	YES	
E152	Basic SET TRANSACTION statement	YES	

ID	Feature	Supported	Comments
E152-01	`ISOLATION LEVEL SERIALIZABLE` clause	NO	Can be declared but is treated as a synonym for `REPEATABLE READ`
E152-02	`READ ONLY` and `READ WRITE` clauses	YES	
E153	Updatable queries with subqueries	NO	
E161	SQL comments using leading double minus	YES	
E171	SQLSTATE support	YES	
E182	Module language	NO	
F021	Basic information schema	YES	
F021-01	`COLUMNS` view	YES	
F021-02	`TABLES` view	YES	
F021-03	`VIEWS` view	YES	
F021-04	`TABLE_CONSTRAINTS` view	YES	
F021-05	`REFERENTIAL_ CONSTRAINTS` view	YES	
F021-06	`CHECK_CONSTRAINTS` view	YES	
F031	Basic schema manipulation	YES	
F031-01	`CREATE TABLE` statement to create persistent base tables	YES	
F031-02	`CREATE VIEW` statement	YES	
F031-03	`GRANT` statement	YES	
F031-04	`ALTER TABLE` statement: `ADD COLUMN` clause	YES	
F031-13	`DROP TABLE` statement: `RESTRICT` clause	YES	
F031-16	`DROP VIEW` statement: `RESTRICT` clause	YES	
F031-19	`REVOKE` statement: `RESTRICT` clause	YES	
F032	`CASCADE` drop behavior	YES	

ID	Feature	Supported	Comments
F033	`ALTER TABLE` statement: `DROP COLUMN` clause	YES	
F034	Extended REVOKE statement	YES	
F034-01	`REVOKE` statement performed by other than the owner of a schema object	YES	
F034-02	`REVOKE` statement: `GRANT OPTION FOR` clause	YES	
F034-03	`REVOKE` statement to revoke a privilege that the grantee has `WITH GRANT OPTION`	YES	
F041	Basic joined table	YES	
F041-01	Inner join (but not necessarily the `INNER` keyword)	YES	
F041-02	`INNER` keyword	YES	
F041-03	`LEFT OUTER JOIN`	YES	
F041-04	`RIGHT OUTER JOIN`	YES	
F041-05	Outer joins can be nested	YES	
F041-07	The inner table in a left or right outer join can also be used in an inner join	YES	
F041-08	All comparison operators are supported (rather than just =)	YES	
F051	Basic date and time	YES	
F051-01	`DATE` data type (including support of `DATE` literal)	YES	
F051-02	`TIME` data type (including support of `TIME` literal) with fractional seconds precision of at least 0	YES	

ID	Feature	Supported	Comments
F051-03	`TIMESTAMP` data type (including support of `TIMESTAMP` literal) with fractional seconds precision of at least 0 and 6	YES	
F051-04	Comparison predicate on `DATE`, `TIME`, and `TIMESTAMP` data types	YES	
F051-05	Explicit `CAST` between datetime types and character string types	YES	
F051-06	`CURRENT_DATE`	YES	
F051-07	`LOCALTIME`	YES	
F051-08	`LOCALTIMESTAMP`	YES	
F052	Intervals and datetime arithmetic	YES	
F053	`OVERLAPS` predicate	YES	
F081	`UNION` and `EXCEPT` in views	YES	
F111	Isolation levels other than SERIALIZABLE	YES	
F111-01	`READ UNCOMMITTED` isolation level	NO	Can be declared but is treated as a synonym for `READ COMMITTED`
F111-02	`READ COMMITTED` isolation level	YES	
F111-03	`REPEATABLE READ` isolation level	YES	
F121	Basic diagnostics management	NO	
F122	Enhanced diagnostics management	NO	
F123	All diagnostics	NO	
F131-	Grouped operations	YES	
F131-01	`WHERE`, `GROUP BY`, and `HAVING` clauses supported in queries with grouped views	YES	
F131-02	Multiple tables supported in queries with grouped views	YES	

ID	Feature	Supported	Comments
F131-03	Set functions supported in queries with grouped views	YES	
F131-04	Subqueries with `GROUP BY` and `HAVING` clauses and grouped views	YES	
F131-05	Single row `SELECT` with `GROUP BY` and `HAVING` clauses and grouped views	YES	
F171	Multiple schemas per user	YES	
F181	Multiple module support	NO	
F191	Referential delete actions	NO	
F200	`TRUNCATE TABLE` statement	YES	
F201	`CAST` function	YES	
F202	`TRUNCATE TABLE`: identity column restart option	NO	
F221	Explicit defaults	YES	
F222	`INSERT` statement: `DEFAULT VALUES` clause	YES	
F231	Privilege tables	YES	
F231-01	`TABLE_PRIVILEGES` view	YES	
F231-02	`COLUMN_PRIVILEGES` view	YES	
F231-03	`USAGE_PRIVILEGES` view	YES	
F251	Domain support		
F261	CASE expression	YES	
F261-01	Simple `CASE`	YES	
F261-02	Searched `CASE`	YES	
F261-03	`NULLIF`	YES	
F261-04	`COALESCE`	YES	
F262	Extended `CASE` expression	NO	

ID	Feature	Supported	Comments
F263	Comma-separated predicates in simple `CASE` expression	NO	
F271	Compound character literals	YES	
F281	`LIKE` enhancements	YES	
F291	`UNIQUE` predicate	NO	
F301	`CORRESPONDING` in query expressions	NO	
F302	`INTERSECT` table operator	YES	
F302-01	`INTERSECT DISTINCT` table operator	YES	
F302-02	`INTERSECT ALL` table operator	YES	
F304	`EXCEPT ALL` table operator		
F311	Schema definition statement	YES	Partial sub-feature support
F311-01	`CREATE SCHEMA`	YES	
F311-02	`CREATE TABLE` for persistent base tables	YES	
F311-03	`CREATE VIEW`	YES	
F311-04	`CREATE VIEW: WITH CHECK OPTION`	NO	
F311-05	`GRANT` statement	YES	
F312	`MERGE` statement	NO	
F313	Enhanced `MERGE` statement	NO	
F321	User authorization	YES	
F341	Usage Tables	NO	
F361	Subprogram support	YES	
F381	Extended schema manipulation	YES	
F381-01	`ALTER TABLE` statement: `ALTER COLUMN` clause		Some limitations on altering distribution key columns
F381-02	`ALTER TABLE` statement: `ADD CONSTRAINT` clause		

ID	Feature	Supported	Comments
F381-03	`ALTER TABLE` statement: `DROP CONSTRAINT` clause		
F382	Alter column data type	YES	Some limitations on altering distribution key columns
F391	Long identifiers	YES	
F392	Unicode escapes in identifiers	NO	
F393	Unicode escapes in literals	NO	
F394	Optional normal form specification	NO	
F401	Extended joined table	YES	
F401-01	`NATURAL JOIN`	YES	
F401-02	`FULL OUTER JOIN`	YES	
F401-04	`CROSS JOIN`	YES	
F402	Named column joins for LOBs, arrays, and multisets	NO	
F403	Partitioned joined tables	NO	
F411	Time zone specification	YES	Differences regarding literal interpretation
F421	National character	YES	
F431	Read-only scrollable cursors	YES	Forward scrolling only
01	`FETCH` with explicit `NEXT`	YES	
02	`FETCH FIRST`	NO	
03	`FETCH LAST`	YES	
04	`FETCH PRIOR`	NO	
05	`FETCH ABSOLUTE`	NO	
06	`FETCH RELATIVE`	NO	
F441	Extended set function support	YES	
F442	Mixed column references in set functions	YES	
F451	Character set definition	NO	
F461	Named character sets	NO	

ID	Feature	Supported	Comments
F471	Scalar subquery values	YES	
F481	Expanded `NULL` predicate	YES	
F491	Constraint management	YES	
F501	Features and conformance views	YES	
F501-01	`SQL_FEATURES` view	YES	
F501-02	`SQL_SIZING` view	YES	
F501-03	`SQL_LANGUAGES` view	YES	
F502	Enhanced documentation tables	YES	
F502-01	`SQL_SIZING_PROFILES` view	YES	
F502-02	`SQL_IMPLEMENTATION_INFO` view	YES	
F502-03	`SQL_PACKAGES` view	YES	
F521	Assertions	NO	
F531	Temporary tables	YES	Non-standard form
F555	Enhanced seconds precision	YES	
F561	Full value expressions	YES	
F571	Truth value tests	YES	
F591	Derived tables	YES	
F611	Indicator data types	YES	
F641	Row and table constructors	NO	
F651	Catalog name qualifiers	YES	
F661	Simple tables	NO	
F671	Subqueries in `CHECK`	NO	Intentionally omitted
F672	Retrospective check constraints	YES	
F690	Collation support	NO	
F692	Enhanced collation support	NO	
F693	SQL-session and client module collations	NO	
F695	Translation support	NO	

ID	Feature	Supported	Comments
F696	Additional translation documentation	NO	
F701	Referential update actions	NO	
F711	`ALTER` domain	YES	
F721	Deferrable constraints	NO	
F731	`INSERT` column privileges	YES	
F741	Referential `MATCH` types	NO	No partial match
F751	View `CHECK` enhancements	NO	
F761	Session management	YES	
F762	`CURRENT_CATALOG`	NO	
F763	`CURRENT_SCHEMA`	NO	
F771	Connection management	YES	
F781	Self-referencing operations	YES	
F791	Insensitive cursors	YES	
F801	Full set function	YES	
F812	Basic flagging	NO	
F813	Extended flagging	NO	
F831	Full cursor update	NO	
F841	`LIKE_REGEX` predicate	NO	Non-standard syntax for regex
F842	`OCCURENCES_REGEX` function	NO	
F843	`POSITION_REGEX` function	NO	
F844	`SUBSTRING_REGEX` function	NO	
F845	`TRANSLATE_REGEX` function	NO	
F846	Octet support in regular expression operators	NO	
F847	Nonconstant regular expressions	NO	
F850	Top-level `ORDER BY` clause in *query expression*	YES	

ID	Feature	Supported	Comments
F851	Top-level `ORDER BY` clause in subqueries	NO	
F852	Top-level `ORDER BY` clause in views	NO	
F855	Nested `ORDER BY` clause in *query expression*	NO	
F856	Nested `FETCH FIRST` clause in *query expression*	NO	
F857	Top-level `FETCH FIRST` clause in *query expression*	NO	
F858	`FETCH FIRST` clause in subqueries	NO	
F859	Top-level `FETCH FIRST` clause in views	NO	
F860	`FETCH FIRST ROW`*count* in `FETCH FIRST` clause	NO	
F861	Top-level `RESULT OFFSET` clause in *query expression*	NO	
F862	`RESULT OFFSET` clause in subqueries	NO	
F863	Nested `RESULT OFFSET` clause in *query expression*	NO	
F864	Top-level `RESULT OFFSET` clause in views	NO	
F865	`OFFSET ROW`*count* in `RESULT OFFSET` clause	NO	
S011	Distinct data types	NO	
S023	Basic structured types	NO	
S024	Enhanced structured types	NO	
S025	Final structured types	NO	
S026	Self-referencing structured types	NO	
S027	Create method by specific method name	NO	
S028	Permutable UDT options list	NO	

ID	Feature	Supported	Comments
S041	Basic reference types	NO	
S043	Enhanced reference types	NO	
S051	Create table of type	NO	
S071	SQL paths in function and type name resolution	YES	
S091	Basic array support	NO	Greenplum has arrays, but is not fully standards compliant
S091-01	Arrays of built-in data types	NO	Partially compliant
S091-02	Arrays of distinct types	NO	
S091-03	Array expressions	NO	
S092	Arrays of user-defined types	NO	
S094	Arrays of reference types	NO	
S095	Array constructors by query	NO	
S096	Optional array bounds	NO	
S097	Array element assignment	NO	

ID	Feature	Supported	Comments
S098	ARRAY_AGG	Partially	Supported: Using array_agg without a window specification; for example SELECT array_agg(x) FROM ... SELECT array_agg (x order by y) FROM ... Not supported: Using array_agg as an aggregate derived window function; for example SELECT array_agg(x) over (ORDER BY y) FROM ... SELECT array_ agg(x order by y) over (PARTITION BY z) FROM ... SELECT array_ agg(x order by y) over (ORDER BY z) FROM ...
S111	ONLY in query expressions	YES	
S151	Type predicate	NO	
S161	Subtype treatment	NO	
S162	Subtype treatment for references	NO	
S201	SQL-invoked routines on arrays	NO	Functions can be passed Greenplum array types
S202	SQL-invoked routines on multisets	NO	
S211	User-defined cast functions	YES	
S231	Structured type locators	NO	
S232	Array locators	NO	
S233	Multiset locators	NO	
S241	Transform functions	NO	
S242	Alter transform statement	NO	
S251	User-defined orderings	NO	

ID	Feature	Supported	Comments
S261	Specific type method	NO	
S271	Basic multiset support	NO	
S272	Multisets of user-defined types	NO	
S274	Multisets of reference types	NO	
S275	Advanced multiset support	NO	
S281	Nested collection types	NO	
S291	Unique constraint on entire row	NO	
S301	Enhanced `UNNEST`	NO	
S401	Distinct types based on array types	NO	
S402	Distinct types based on distinct types	NO	
S403	`MAX_CARDINALITY`	NO	
S404	`TRIM_ARRAY`	NO	
T011	Timestamp in Information Schema	NO	
T021	`BINARY` and `VARBINARY` data types	NO	
T022	Advanced support for `BINARY` and `VARBINARY` data types	NO	
T023	Compound binary literal	NO	
T024	Spaces in binary literals	NO	
T031	`BOOLEAN` data type	YES	
T041	Basic `LOB` data type support	NO	
T042	Extended `LOB` data type support	NO	
T043	Multiplier T	NO	
T044	Multiplier P	NO	
T051	Row types	NO	
T052	`MAX` and `MIN` for row types	NO	
T053	Explicit aliases for all-fields reference	NO	

ID	Feature	Supported	Comments
T061	UCS support	NO	
T071	`BIGINT` data type	YES	
T101	Enhanced nullability determiniation	NO	
T111	Updatable joins, unions, and columns	NO	
T121	`WITH` (excluding `RECURSIVE`) in query expression	NO	
T122	`WITH` (excluding `RECURSIVE`) in subquery	NO	
T131	Recursive query	NO	
T132	Recursive query in subquery	NO	
T141	`SIMILAR` predicate	YES	
T151	`DISTINCT` predicate	YES	
T152	`DISTINCT` predicate with negation	NO	
T171	`LIKE` clause in table definition	YES	
T172	AS subquery clause in table definition	YES	
T173	Extended `LIKE` clause in table definition	YES	
T174	Identity columns	NO	
T175	Generated columns	NO	
T176	Sequence generator support	NO	
T177	Sequence generator support: simple restart option	NO	
T178	Identity columns: simple restart option	NO	
T191	Referential action `RESTRICT`	NO	
T201	Comparable data types for referential constraints	NO	
T211	Basic trigger capability	NO	

ID	Feature	Supported	Comments
T211-01	Triggers activated on `UPDATE`, `INSERT`, or `DELETE` of one base table	NO	
T211-02	`BEFORE` triggers	NO	
T211-03	`AFTER` triggers	NO	
T211-04	`FOR EACH ROW` triggers	NO	
T211-05	Ability to specify a search condition that must be true before the trigger is invoked	NO	
T211-06	Support for run-time rules for the interaction of triggers and constraints	NO	
T211-07	`TRIGGER` privilege	YES	
T211-08	Multiple triggers for the same event are executed in the order in which they were created in the catalog	NO	Intentionally omitted
T212	Enhanced trigger capability	NO	
T213	`INSTEAD OF` triggers	NO	
T231	Sensitive cursors	YES	
T241	`START TRANSACTION` statement	YES	
T251	`SET TRANSACTION` statement: `LOCAL` option	NO	
T261	Chained transactions	NO	
T271	Savepoints	YES	
T272	Enhanced savepoint management	NO	
T281	`SELECT` privilege with column granularity	YES	
T285	Enhanced derived column names	NO	
T301	Functional dependencies	NO	
T312	OVERLAY function	YES	
T321	Basic SQL-invoked routines	NO	Partial support

ID	Feature	Supported	Comments
T321-01	User-defined functions with no overloading	YES	
T321-02	User-defined stored procedures with no overloading	NO	
T321-03	Function invocation	YES	
T321-04	CALL statement	NO	
T321-05	RETURN statement	NO	
T321-06	ROUTINES view	YES	
T321-07	PARAMETERS view	YES	
T322	Overloading of SQL-invoked functions and procedures	YES	
T323	Explicit security for external routines	YES	
T324	Explicit security for SQL routines	NO	
T325	Qualified SQL parameter references	NO	
T326	Table functions	NO	
T331	Basic roles	NO	
T332	Extended roles	NO	
T351	Bracketed SQL comments (`/*...*/` comments)	YES	
T431	Extended grouping capabilities	NO	
T432	Nested and concatenated `GROUPING SETS`	NO	
T433	Multiargument `GROUPING` function	NO	
T434	`GROUP BY DISTINCT`	NO	
T441	`ABS` and `MOD` functions	YES	
T461	Symmetric `BETWEEN` predicate	YES	
T471	Result sets return value	NO	
T491	`LATERAL` derived table	NO	
T501	Enhanced `EXISTS` predicate	NO	

ID	Feature	Supported	Comments
T511	Transaction counts	NO	
T541	Updatable table references	NO	
T561	Holdable locators	NO	
T571	Array-returning external SQL-invoked functions	NO	
T572	Multiset-returning external SQL-invoked functions	NO	
T581	Regular expression substring function	YES	
T591	`UNIQUE` constraints of possibly null columns	YES	
T601	Local cursor references	NO	
T611	Elementary OLAP operations	YES	
T612	Advanced OLAP operations	NO	Partially supported
T613	Sampling	NO	
T614	`NTILE` function	YES	
T615	`LEAD` and `LAG` functions	YES	
T616	Null treatment option for `LEAD` and `LAG` functions	NO	
T617	`FIRST_VALUE` and `LAST_VALUE` function	YES	
T618	`NTH_VALUE`	NO	Function exists in Greenplum but not all options are supported
T621	Enhanced numeric functions	YES	
T631	N predicate with one list element	NO	
T641	Multiple column assignment	NO	Some syntax variants supported
T651	SQL-schema statements in SQL routines	NO	
T652	SQL-dynamic statements in SQL routines	NO	
T653	SQL-schema statements in external routines	NO	

ID	Feature	Supported	Comments
T654	SQL-dynamic statements in external routines	NO	
T655	Cyclically dependent routines	NO	
M001	Datalinks	NO	
M002	Datalinks via SQL/CLI	NO	
M003	Datalinks via Embedded SQL	NO	
M004	Foreign data support	NO	
M005	Foreign schema support	NO	
M006	GetSQLString routine	NO	
M007	TransmitRequest	NO	
M009	GetOpts and GetStatistics routines	NO	
M010	Foreign data wrapper support	NO	
M011	Datalinks via Ada	NO	
M012	Datalinks via C	NO	
M013	Datalinks via COBOL	NO	
M014	Datalinks via Fortran	NO	
M015	Datalinks via M	NO	
M016	Datalinks via Pascal	NO	
M017	Datalinks via PL/I	NO	
M018	Foreign data wrapper interface routines in Ada	NO	
M019	Foreign data wrapper interface routines in C	NO	
M020	Foreign data wrapper interface routines in COBOL	NO	
M021	Foreign data wrapper interface routines in Fortran	NO	
M022	Foreign data wrapper interface routines in MUMPS	NO	
M023	Foreign data wrapper interface routines in Pascal	NO	

ID	Feature	Supported	Comments
M024	Foreign data wrapper interface routines in PL/I	NO	
M030	SQL-server foreign data support	NO	
M031	Foreign data wrapper general routines	NO	
X010	XML type	YES	
X011	Arrays of XML type	YES	
X012	Multisets of XML type	NO	
X013	Distinct types of XML type	NO	
X014	Attributes of XML type	NO	
X015	Fields of XML type	NO	
X016	Persistent XML values	YES	
X020	XMLConcat	YES	xmlconcat2() supported
X025	XMLCast	NO	
X030	XMLDocument	NO	
X031	XMLElement	YES	
X032	XMLForest	YES	
X034	XMLAgg	YES	
X035	XMLAgg: ORDER BY option	YES	
X036	XMLComment	YES	
X037	XMLPI	YES	
X038	XMLText	NO	
X040	Basic table mapping	NO	
X041	Basic table mapping: nulls absent	NO	
X042	Basic table mapping: null as nil	NO	
X043	Basic table mapping: table as forest	NO	
X044	Basic table mapping: table as element	NO	
X045	Basic table mapping: with target namespace	NO	
X046	Basic table mapping: data mapping	NO	

ID	Feature	Supported	Comments
X047	Basic table mapping: metadata mapping	NO	
X048	Basic table mapping: base64 encoding of binary strings	NO	
X049	Basic table mapping: hex encoding of binary strings	NO	
X051	Advanced table mapping: nulls absent	NO	
X052	Advanced table mapping: null as nil	NO	
X053	Advanced table mapping: table as forest	NO	
X054	Advanced table mapping: table as element	NO	
X055	Advanced table mapping: target namespace	NO	
X056	Advanced table mapping: data mapping	NO	
X057	Advanced table mapping: metadata mapping	NO	
X058	Advanced table mapping: base64 encoding of binary strings	NO	
X059	Advanced table mapping: hex encoding of binary strings	NO	
X060	XMLParse: Character string input and CONTENT option	YES	
X061	XMLParse: Character string input and DOCUMENT option	YES	
X065	XMLParse: BLOB input and CONTENT option	NO	
X066	XMLParse: BLOB input and DOCUMENT option	NO	
X068	XMLSerialize: BOM	NO	
X069	XMLSerialize: INDENT	NO	

ID	Feature	Supported	Comments
X070	XMLSerialize: Character string serialization and CONTENT option	YES	
X071	XMLSerialize: Character string serialization and DOCUMENT option	YES	
X072	XMLSerialize: Character string serialization	YES	
X073	XMLSerialize: BLOB serialization and CONTENT option	NO	
X074	XMLSerialize: BLOB serialization and DOCUMENT option	NO	
X075	XMLSerialize: BLOB serialization	NO	
X076	XMLSerialize: VERSION	NO	
X077	XMLSerialize: explicit ENCODING option	NO	
X078	XMLSerialize: explicit XML declaration	NO	
X080	Namespaces in XML publishing	NO	
X081	Query-level XML namespace declarations	NO	
X082	XML namespace declarations in DML	NO	
X083	XML namespace declarations in DDL	NO	
X084	XML namespace declarations in compound statements	NO	
X085	Predefined namespace prefixes	NO	
X086	XML namespace declarations in XMLTable	NO	
X090	XML document predicate	NO	xml_is_well_formed_ document() supported
X091	XML content predicate	NO	xml_is_well_formed_ content() supported
X096	XMLExists	NO	xmlexists() supported

ID	Feature	Supported	Comments
X100	Host language support for XML: CONTENT option	NO	
X101	Host language support for XML: DOCUMENT option	NO	
X110	Host language support for XML: VARCHAR mapping	NO	
X111	Host language support for XML: CLOB mapping	NO	
X112	Host language support for XML: BLOB mapping	NO	
X113	Host language support for XML: STRIP WHITESPACE option	YES	
X114	Host language support for XML: PRESERVE WHITESPACE option	YES	
X120	XML parameters in SQL routines	YES	
X121	XML parameters in external routines	YES	
X131	Query-level XMLBINARY clause	NO	
X132	XMLBINARY clause in DML	NO	
X133	XMLBINARY clause in DDL	NO	
X134	XMLBINARY clause in compound statements	NO	
X135	XMLBINARY clause in subqueries	NO	
X141	IS VALID predicate: data-driven case	NO	
X142	IS VALID predicate: ACCORDING TO clause	NO	
X143	IS VALID predicate: ELEMENT clause	NO	
X144	IS VALID predicate: schema location	NO	

ID	Feature	Supported	Comments
X145	IS VALID predicate outside check constraints	NO	
X151	IS VALID predicate with DOCUMENT option	NO	
X152	IS VALID predicate with CONTENT option	NO	
X153	IS VALID predicate with SEQUENCE option	NO	
X155	IS VALID predicate: NAMESPACE without ELEMENT clause	NO	
X157	IS VALID predicate: NO NAMESPACE with ELEMENT clause	NO	
X160	Basic Information Schema for registered XML Schemas	NO	
X161	Advanced Information Schema for registered XML Schemas	NO	
X170	XML null handling options	NO	
X171	NIL ON NO CONTENT option	NO	
X181	XML(DOCUMENT (UNTYPED)) type	NO	
X182	XML(DOCUMENT (ANY)) type	NO	
X190	XML(SEQUENCE) type	NO	
X191	XML(DOCUMENT (XMLSCHEMA)) type	NO	
X192	XML(CONTENT (XMLSCHEMA)) type	NO	
X200	XMLQuery	NO	
X201	XMLQuery: RETURNING CONTENT	NO	
X202	XMLQuery: RETURNING SEQUENCE	NO	
X203	XMLQuery: passing a context item	NO	

ID	Feature	Supported	Comments
X204	XMLQuery: initializing an XQuery variable	NO	
X205	XMLQuery: EMPTY ON EMPTY option	NO	
X206	XMLQuery: NULL ON EMPTY option	NO	
X211	XML 1.1 support	NO	
X221	XML passing mechanism BY VALUE	NO	
X222	XML passing mechanism BY REF	NO	
X231	XML(CONTENT (UNTYPED)) type	NO	
X232	XML(CONTENT (ANY)) type	NO	
X241	RETURNING CONTENT in XML publishing	NO	
X242	RETURNING SEQUENCE in XML publishing	NO	
X251	Persistent XML values of XML(DOCUMENT (UNTYPED)) type	NO	
X252	Persistent XML values of XML(DOCUMENT (ANY)) type	NO	
X253	Persistent XML values of XML(CONTENT (UNTYPED)) type	NO	
X254	Persistent XML values of XML(CONTENT (ANY)) type	NO	
X255	Persistent XML values of XML(SEQUENCE) type	NO	
X256	Persistent XML values of XML(DOCUMENT (XMLSCHEMA)) type	NO	
X257	Persistent XML values of XML(CONTENT (XMLSCHEMA) type	NO	
X260	XML type: ELEMENT clause	NO	

ID	Feature	Supported	Comments
X261	XML type: NAMESPACE without ELEMENT clause	NO	
X263	XML type: NO NAMESPACE with ELEMENT clause	NO	
X264	XML type: schema location	NO	
X271	XMLValidate: data-driven case	NO	
X272	XMLValidate: ACCORDING TO clause	NO	
X273	XMLValidate: ELEMENT clause	NO	
X274	XMLValidate: schema location	NO	
X281	XMLValidate: with DOCUMENT option	NO	
X282	XMLValidate with CONTENT option	NO	
X283	XMLValidate with SEQUENCE option	NO	
X284	XMLValidate NAMESPACE without ELEMENT clause	NO	
X286	XMLValidate: NO NAMESPACE with ELEMENT clause	NO	
X300	XMLTable	NO	
X301	XMLTable: derived column list option	NO	
X302	XMLTable: ordinality column option	NO	
X303	XMLTable: column default option	NO	
X304	XMLTable: passing a context item	NO	
X305	XMLTable: initializing an XQuery variable	NO	
X400	Name and identifier mapping	NO	

Chapter 8

Greenplum Client and Loader Tools Package

This documentation describes the contents of, and how to install, configure, and use the Greenplum client and loader utility programs for UNIX and Windows systems.

Key topics in the documentation include:

- *About the Tools Package*
- *Installing the Client and Loader Tools Package*
- *Configuring Greenplum Database for Remote Client Access*
- *Configuring a Client System for Kerberos Authentication*
- *Using the Client and Loader Tools*
- *Client and Loader Utility Reference*

Chapter 9

About the Tools Package

The Greenplum client and loader tools package provides utility programs that you can install and run on a host outside of your Greenplum Database cluster. The package is available on *VMware Tanzu Network*.

Greenplum utility programs provided in the client and loader tools package include:

Table 263: Client and Loader Programs

Program	Description
createdb	Create a Greenplum database. Requires superuser or specially-granted Greenplum Database privileges.
createlang	Register a language in a Greenplum database. Requires superuser or specially-granted Greenplum Database privileges.
createuser	Register a Greenplum user. Requires superuser or specially-granted Greenplum Database privileges.
dropdb	Drop a Greenplum database. Requires superuser or specially-granted Greenplum Database privileges.
droplang	Remove support for a language from a Greenplum database. Requires superuser or specially-granted Greenplum Database privileges.
dropuser	Remove a Greenplum user. Requires superuser or specially-granted Greenplum Database privileges.
gpfdist	Greenplum parallel file distribution program.
gpkafka	Load Kafka data into Greenplum Database using the Pivotal Greenplum Streaming Server.
gpload	Greenplum data loading utility.
gpss	Start a Pivotal Greenplum Streaming Server instance.
gpsscli	Pivotal Greenplum Streaming Server client program.
pg_dump	Dump the contents of a Greenplum database to a file.
pg_dumpall	Dump the contents of all Greenplum databases to a file.
psql	PostgreSQL interactive command-line interface for Greenplum Database.

Note: The Windows Greenplum client and loader tools package provides additional libraries and programs, including `kinit`, `kdestroy`, and `klist`. The Windows package does **not** include the `gpkafka`, `gpsscli`, and `gpss` programs.

The `gpload` program provided in the Windows package is backwards-compatible with Greenplum Database 5.

Chapter 10

Installing the Client and Loader Tools Package

This section provides the information required to download and install the tools on your client machine.

Supported Platforms

You can install the client and loader tools package on any of the following systems:

- CentOS 6.x
- CentOS 7.x
- Ubuntu 18.04
- Windows 7 SP1 or later
- Windows Server 2012 or later

Installation Procedure

Perform the following procedure to install the Greenplum Database client tools on your system:

1. Download the appropriate installer package for your platform from *VMware Tanzu Network*. For example, to download the CentOS 7.x package, click to select the **Greenplum Clients->Clients for RHEL 7** directory.

   The naming format of a UNIX package file is `greenplum-db-clients-<version>-<platform>.<filetype>`.

   The naming format of a Windows package file is `greenplum-db-clients-<version>-x86_64.msi`.
2. *Note the file system location of the downloaded file.*
3. If you are installing the package on a system running the CentOS or Ubuntu operating system, follow the instructions in *Running the UNIX Tools Installer*.
4. If you are installing the package on a Windows system, follow the instructions in *Running the Windows Tools Installer*.

About Your Installation

Your Greenplum Database client and loader tools installation includes the following files and directories:

Table 264: Files and Directories

File/Directory	Description
bin/	client and loader tool utility programs
ext/ (UNIX only)	Python runtime components required by the UNIX utilities
greenplum_clients_path.<ext>	environment set up script or batch file; the file extension (<ext>) is operating system- or shell-dependent
lib/	libraries required by the utilities
LICENSE (UNIX) 'Pivotal License' (Windows)	license notices for the utilities
NOTICE (UNIX) 'Thirdparty Notice' (Windows)	attribution notices for the utilities
'GPDB Clients Version' (Windows only)	file identifying the Windows package version

Running the UNIX Tools Installer

This section describes the client and loader tool package installation procedure for CentOS and Ubuntu systems.

Prerequisites

You must have operating system superuser privileges to install the tools package.

> **Note:** Installing the client tools package automatically installs dependent packages not already installed on the system.

Procedure

Perform the following procedure to install the client and loader tools package on a CentOS or Ubuntu system.

1. Locate the installer file that you downloaded from VMware Tanzu Network. The naming format of the file is `greenplum-db-clients-<version>-<platform>.<file_type>`.
2. Install the package using your package management utility. You must be the superuser or have `sudo` access to install packages. For example:
   - To install the tools on a host running CentOS 7.x:

     ```
     root@clientsys$ yum install greenplum-db-clients-6.1.0-rhel7-x86_64.rpm
     ```

   - To install the tools on a host running Ubuntu 18.04:

     ```
     root@clientsys$ apt install greenplum-db-clients-6.1.0-ubuntu18.04-
     amd64.deb
     ```

   The client tools are installed into the `/usr/local/greenplum-db-clients-<version>/` directory. The installation process creates a symbolic link from `/usr/local/greenplum-db-clients` to the install directory.

Running the Windows Tools Installer

This section describes the client and loader tool package installation procedure for Windows systems.

Prerequisites

You must have operating system superuser privileges to install the tools package.

Procedure

Perform the following procedure to install the client and loader tools package on a Windows system.

1. The Greenplum Database client and loader tools for Windows require a recent Microsoft Visual C++ Redistributable for Visual Studio 2017. You must download and install an update as described in the Microsoft support article titled *The latest supported Visual C++ downloads*.
2. If you plan to use the `gpload.bat` Greenplum Database loader program for Windows:
   a. Ensure that a 64-bit version of Python 2.7 is installed on your system. Refer to *Python 2.7.16* or the source of your choice for Python download and install instructions.
   b. You must also add the Python directory to your `PATH`.
3. Locate the installer `.msi` file that you downloaded from VMware Tanzu Network in a previous step. The naming format of the Windows installer file is `greenplum-db-clients-<version>-x86_64.msi`.
4. Double-click on the `greenplum-db-clients-<version>-x86_64.msi` file to launch the installer.
5. Click **Next** on the **Greenplum Clients Setup Wizard** Welcome screen.
6. Read through the **End-User License Agreement**, and click **I Agree** to accept the terms of the license.
7. By default, the Greenplum Database client and load tools are installed into the following directory:

   ```
   C:\Program Files\Greenplum\greenplum-clients\
   ```

   Click **Browse** on the **Custom Setup** screen to choose another location.
8. Click **Next** when you have chosen the desired install path.
9. Click **Install** to begin the installation.
10. Click **Finish** to exit the Windows client and load tools installer.

Chapter 11

Configuring Greenplum Database for Remote Client Access

Greenplum Database does not, by default, accept remote client connections. You must configure Greenplum Database to accept remote connections. This configuration involves identifying each client host system and Greenplum Database role combination to which you want to provide access, and then adding access rules to the `pg_hba.conf` client authentication configuration file. Refer to *Allowing Connections to Greenplum Database* for detailed information about configuring remote client access.

> **Note:** Ensure that the authentication method that you configure for a role is supported by the Greenplum Database client program(s) that the role will execute.

In addition to configuring remote client access, you must also ensure that each Greenplum role that you are allowing to connect to the master exists in the cluster, and that the role has the correct privileges to database objects. *Managing Roles and Privileges* describes configuring Greenplum Database users and granting privileges.

Chapter 12

Configuring a Client System for Kerberos Authentication

If your Greenplum Database cluster employs Kerberos user authentication, your client host must be configured to access Greenplum with Kerberos. Refer to the following documentation for instructions on configuring Kerberos authentication on a client system:

- UNIX client hosts - *Configuring Kerberos for Linux Clients*
- Windows client hosts - *Configuring Kerberos for Windows Clients*

If your Greenplum Database cluster is not using Kerberos for user authentication, then this configuration is not required.

Chapter 13

Using the Client and Loader Tools

This section provides the information required to set up your Greenplum Database client runtime environment and use the client and loader tools. Topics include:

- *Prerequisites*
- *Setting Up Your Greenplum Database Clients Runtime Environment*
- *Running the Client and Loader Programs*
- *Greenplum Database Documentation References*
- *Windows Considerations*

Prerequisites

Before using the client and loader tools, ensure that:

- Your Greenplum Database cluster is up and running, and you can identify the master host and port number, if the master server process is not running on the default port (5432).
- Network connectivity exists between the client machine and the Greenplum Database master host. If you are using the `gpfdist`, `gpload`, or `gpss` utility programs, network connectivity must also exist between the client machine and all Greenplum Database segment hosts.
- You have installed and configured the tools and any dependent components on your client machine as described in *Installing the Client and Loader Tools Package*.
- You can identify your Greenplum Database user/role name and password.
- You have created or can identify the Greenplum database, schema, and table objects of interest.

Contact your Greenplum Database administrator if you do not meet the prerequisites mentioned above.

Setting Up Your Greenplum Database Clients Runtime Environment

The client and loader tools package installs a file that you use to set up your Greenplum Database client and loader environment. This script or batch file, named `greenplum_clients_path.<ext>` (where the file extension `<ext>` is operating system- or shell-dependent) is located in the client tools root install directory.

`greenplum_clients_path.<ext>`:

- Sets the runtime environment variables that are required by the utilities.
- Sets the `$GPHOME_CLIENTS` environment variable to point to the root directory of the client and loader tools installation.
- Updates your `$PATH` to include `$GPHOME_CLIENTS/bin`.

You must source or run `greenplum_clients_path.<ext>` before you invoke any of the client or loader programs. For example, run the following command on a CentOS or Ubuntu system to source the file:

```
user@clientsys$ . /usr/local/greenplum-db-clients/greenplum_clients_path.sh
```

> **Note:** Consider adding the command to source or run `greenplum_clients_path.<ext>` to your shell or equivalent initialization file.

Running the Client and Loader Programs

Clients always connect to Greenplum Database through the master host. In order for the client or loader program to establish a connection to the master host, you provide the following connection parameters to the program via options, a configuration file, or environment variables:

Table 265: Connection Parameters

Connection Parameter	Description	Environment Variable Name
Database name	The name of the database to which you want to connect.	PGDATABASE
Host name	The host name of the Greenplum Database master. The default host is the local host.	PGHOST
Port	The port number on which the Greenplum Database master server instance is running. The default port is 5432.	PGPORT
User name	The Greenplum Database user (role) name. This name may not necessarily be the same as your operating system user name.	PGUSER

Note: Refer to the *Client and Loader Utility Reference* for the client or load command to determine tool support for specifying these connection parameters via options, configuration property names, and/or environment variables.

Greenplum Database Documentation References

The following Greenplum Database documentation topics provide additional information about using selected client and loader tools:

- `gpfdist` - *Using the Greenplum Parallel File Server (gpfdist)*
- `gpkafka` - Pivotal Greenplum Streaming Server *Loading from a Kafka data source* documentation
- `gpload` - *Loading Data with gpload*
- `gpss`, `gpsscli` - *Pivotal Greenplum Streaming Server* documentation
- `psql` - *Connecting with psql*

Windows Considerations

Keep in mind these additional considerations when you use the Windows client and load programs:

- You must ensure that any ports that you identify in a `gpload` control file are unblocked by any firewall running on the Windows client system.
- By default, `gpload.bat` attempts to create a directory named `gpAdminLog` in the directory from which you execute the program, and writes its log files there. This operation will fail if you do not have write permission to the current working directory. Run `gpload.bat` with the `-l` option to direct the log output to a different location.
- Review the Greenplum *Character Encoding* and *Formatting Rows* documentation for Windows-specific considerations for the tools.

Chapter 14

Client and Loader Utility Reference

The Greenplum client and loader tools package includes the following utilities:

Note: The Windows Greenplum client and loader tools package provides additional libraries and programs, including `kinit`, `kdestroy`, and `klist`. The Windows package does **not** include the `gpkafka`, `gpsscli`, and `gpss` programs.

The `gpload` program provided in the Windows package is backwards-compatible with Greenplum Database 5.

Chapter 15

DataDirect ODBC Drivers for Greenplum

ODBC drivers enable third party applications to connect via a common interface to the Greenplum Database system. This document describes how to install DataDirect Connect XE for ODBC drivers for Greenplum on either a Linux or Windows system. Unless specified otherwise, references to DataDirect Connect XE for ODBC refer to DataDirect Connect XE for ODBC and DataDirect Connect64 XE for ODBC.

The DataDirect ODBC Drivers for Greenplum are available for download from *VMware Tanzu Network*.

Prerequisites

- Install KornShell (ksh) on your system if it is not available.
- Note the appropriate serial number and license key (use the same number for both the serial number and license key during the installation):

Driver	Serial Number / License Key
DataDirect Connect XE for ODBC 7.1 drivers (32-bit drivers)	1076681728
DataDirect Connect64 XE for ODBC 7.1 drivers (64-bit drivers)	1076681984

Supported Client Platforms

DataDirect Connect64 XE for ODBC drivers for Greenplum support the following 64-bit client platforms:

- AIX 64: 7.1, 6.1, 5.3 Fixpack 5 or higher
- HP-UX IPF: 11i v3.0 (B.11.3X), 11i v2.0 (B.11.23)
- Linux Itanium: Red Hat Enterprise Linux (RHEL) 7.x, 6.x, RHEL 5.x, RHEL 4.x
- Linux x64: RHEL 7.x RHEL 6.x, RHEL 5.x, RHEL 4.x, SUSE Linux Enterprise Server (SLES) 15, SLES 12, SLES 11, SLES 10, Ubuntu 16.04
- Solaris on SPARC: 11 and 11 Express (Solaris 5.11), 10 (Solaris 5.10), 9 (Solaris 5.9), 8 (Solaris 5.8)
- Solaris x64: 11 (Solaris 5.11), 10 (Solaris 5.10)
- Windows x64: Windows 8, Windows 10, Windows Server 20016

DataDirect Connect XE for ODBC drivers for Greenplum support the following 32-bit client platforms:

- AIX 32: 7.1, 6.1, 5.3 Fixpack 5 or higher
- HP-UX IPF: 11i v3.0 (B.11.3X), 11i v2.0 (B.11.23)
- HP-UX PA-RISC: 11i v3 (B.11.3X), 11i v2 (B.11.23) 11i v1 (B.11.11), 11
- Linux x86: Red Hat Enterprise Linux (RHEL) 6.x, RHEL 5.x, RHEL 4.x, SUSE Linux Enterprise Server (SLES) 11, SLES 10, Ubuntu 16.04, Ubuntu 14.04
- Solaris on SPARC: 11 and 11 Express (Solaris 5.11), 10 (Solaris 5.10), 9 (Solaris 5.9), 8 (Solaris 5.8)
- Windows: Windows 8, Windows 10, Windows Server 20016

Installing on Linux Systems

To install ODBC drivers on your client:

1. Log into *VMware Tanzu Network* and download the correct ODBC driver for your operating system. The following Linux and UNIX files are available:
   - PROGRESS_DATADIRECT_CONNECT64_ODBC_7.1.6.HOTFIX_LINUX_64.tar.gz
   - PROGRESS_DATADIRECT_CONNECT_ODBC_7.1.6.HOTFIX_LINUX_32.tar.gz
   - PROGRESS_DATADIRECT_CONNECT64_ODBC_7.1.6.HOTFIX_AIX_64.tar.gz
   - PROGRESS_DATADIRECT_CONNECT_ODBC_7.1.6.HOTFIX_AIX_32.tar.gz
2. Unpack the files. For example:

   ```
   $ tar -zxvf
    PROGRESS_DATADIRECT_CONNECT64_ODBC_7.1.6.HOTFIX_LINUX_64.tar.gz
   ```

   The files are extracted to the current directory.
3. Execute the installer as the root user:

   ```
   $ ksh unixmi.ksh
   Progress DataDirect Connect for ODBC Setup is preparing....

   English has been set as the installation language.

   Log file : /tmp/logfile.492.1
   ------------------------------------------------------------------------
   Progress DataDirect Connect (R) and Connect XE for ODBC 7.1 SP5
   for UNIX operating systems
   ------------------------------------------------------------------------

   The following operating system has been detected:

   LinuxX64
   Is this the current operating system on your machine (Y/N) ?
   ```
4. Press Y to confirm your operating system. The installer displays the license agreement.
5. Enter YES to accept the End User License Agreement. The installer prompts you for registration information:

   ```
   Enter YES to accept the above agreement : YES
   Please enter the following information for proper registration.

   In the Key field, enter either EVAL or the Key provided.

   Name          :
   ```
6. Enter the required registration information at each prompt:

Prompt	Enter
Name:	Name to associate with the registration.
Company:	Your company name.
Serial Number:	• 1076681984 for 64-bit driver, or • 1076681728 for 32-bit driver.

Prompt	Enter
Key:	• 1076681984 for 64-bit driver, or • 1076681728 for 32-bit driver.

The installation program displays the registered driver information. For example:

```
You have chosen the Greenplum Wire Protocol driver.

Server Unlimited
Unlimited Connections

To change this information, enter C. Otherwise, press Enter to continue. :
```

7. Press Enter to continue with the installation. The installer prompts you for a temporary directory:

```
DataDirect Connect for ODBC Setup is preparing the installation.
Choose a temporary directory.

Enter the full path to the temporary install directory.[/tmp]:
```

8. Press Enter to accept the default `/tmp` directory or enter a custom directory to store temporary files. The installer extracts temporary files and prompts you for an installation directory:

```
Checking for available space...

There is enough space.
Extracting files...

Choose a destination directory.
Enter the full path to the install directory.[/opt/Progress/DataDirect/
Connect64_for_ODBC_71]:
```

9. Press Enter to accept the default directory or enter a custom destination directory. The installer checks for available space and installs the software:

```
Checking for available space...

There is enough space.
Extracting files...

Creating license file......


DataDirect Connect for ODBC Setup successfully removed all of the
 temporary files.


Thank you for using Progress DataDirect products under OEM license to
 Greenplum Inc.

Would you like to install another product (Y/N) ? [Y]
```

10. Enter `N` to exit the installer.

Configuring the Driver on Linux

After you install the driver software, perform these steps to configure the driver.

1. Change to the installation directory for your driver. For example:

```
$ cd /opt/Progress/DataDirect/Connect64_for_ODBC_71/
```

2. Set the `LD_LIBRARY_PATH`, `ODBCINI` and `ODBCINST` environment variables with the command:

```
$ source odbc.sh
```

3. Open the `odbc.ini` file and create a new DSN entry. You can use the existing "Greenplum Wire Protocol" entry as a template.

```
$ vi $ODBCINI
```

You must edit the following entries to add values that match your system:

Entry	Description
Database	Greenplum database name.
HostName	Master host name.
PortNumber	Master host port number.
LogonID	Greenplum Database user.
Password	Password.

4. Verify the driver version:

```
$ cd /opt/Progress/DataDirect/Connect64_for_ODBC_71/bin
$ ./ddtestlib ddgplm27.so
Load of ddgplm27.so successful, qehandle is 0x15C9EC0
File version: 07.16.0389 (B0562, U0408)
```

Testing the Driver Connection on Linux

To test the DSN connection:

1. Execute the example utility to test the DSN connection, entering the Greenplum Wire Protocol data source name and the credentials of a Greenplum user. For example:

```
$ cd /opt/Progress/DataDirect/Connect64_for_ODBC_71/samples/example
$ ./example
./example DataDirect Technologies, Inc. ODBC Example Application.
Enter the data source name : Greenplum Wire Protocol
Enter the user name        : gpadmin
Enter the password         : gpadmin

Enter SQL statements (Press ENTER to QUIT)
SQL>
```

2. Enter the following select statement to confirm database connectivity:

```
Enter SQL statements (Press ENTER to QUIT)
SQL> select version();

version
PostgreSQL 8.3.23 (Greenplum Database 5.0.0 build
 commit:8c709516061cff5476c03d6e2da99aae42722ae1) on x86_64-pc-linux-gnu,
 compiled by GCC gcc (GCC) 6.2.0 compiled on Sep  1 2017 22:39:53


Enter SQL statements (Press ENTER to QUIT)
```

```
SQL>
```

3. Press the ENTER key to exit the example application.

Installing on Windows Systems

To install ODBC drivers on your client:

1. Log into *VMware Tanzu Network* and download the correct ODBC driver for your operating system (32-bit or 64-bit). The following Windows files are available:
   - `PROGRESS_DATADIRECT_CONNECT64_ODBC_7.1.6.HOTFIX_WIN_64.zip`
   - `PROGRESS_DATADIRECT_CONNECT_ODBC_7.1.6.HOTFIX_WIN_32.zip`
2. Uncompress the installer.
3. Double-click `setup.exe` to launch the install wizard.
4. If necessary, permit the InstallAnywhere installer to run.
5. Click **Next** at the Introduction screen to begin the installation.
6. Accept the End User License Agreement and click **Next**.
7. Select **OEM or Licensed Installation** as the installation type and click **Next**.
8. Enter your licensing information: Division name, Company Name, and serial number/license key found in *Prerequisites*.
9. Select **Add**. You should see this driver in the License dialog box: ODBC Greenplum Wire Protocol Third Party All Platform Server Unlimited Cores
10. Select **Next**.
11. Choose options appropriate for your installation. For example, select to replace the existing drivers and/or to create the default data sources. Click **Next**.
12. Accept the default installation directory or choose a custom directory. Click **Next**.
13. Verify the selected installation options, and click **Install** to begin installation. The installation process may take several minutes.
14. Select **Done** to complete installing the driver package.

Verifying the Version on Windows

To verify your driver version:

1. Select **Start > All Programs > DataDirect > ODBC Administrator** to open the Windows ODBC Administrator.
2. Click the **Drivers** tab, and scroll down to DataDirect <version> Greenplum Wire Protocol. Ensure that you see the expected version number.

Configuring and Testing the Driver on Windows

To configure and test a DSN connection to a Greenplum Database:

1. Open the ODBC Administrator.
2. Select the **System DSN** tab.
3. Select **Add**.
4. Select **DataDirect 7.1 Greenplum Wire Protocol** and click **Finish**.
5. Enter the details for your chosen Greenplum Database instance. For example:

   Recommended: Set the Max Long Varchar size.

   Select the **Advanced** tab.

   In **Max Long Varchar Size**, enter `8192` then select **Apply**.

6. Select **Test Connect**.
7. Enter your user name and password, then select **OK**.
8. You should see the confirmation message **Connection Established!**

If your connection fails, check the following for accuracy:

- Host Name
- Port Number
- Database Name
- User Name
- Password
- Greenplum instance is active

DataDirect Driver Documentation

For more information on working with Data Direct, see documentation that is installed with the driver. By default, you can access the installed documentation by using a Web browser to open the file `/opt/Progress/DataDirect/Connect64_for_ODBC_71/help/index.html`.

Documentation is also available online at *https://www.progress.com/documentation/datadirect-connectors*. Titles include:

- *User's Guide*
- *Reference*
- *Troubleshooting Guide*
- *Installation Help*
- *Windows Readme*
- *UNIX/Linux Readme*

Chapter 16

DataDirect JDBC Driver for Greenplum

DataDirect JDBC drivers are compliant with the Type 4 architecture, but provide advanced features that define them as Type 5 drivers. Additionally, the drivers consistently support the latest database features and are fully compliant with Java™ SE 8 and JDBC 4.0 functionality.

The DataDirect JDBC Driver for Greenplum is available for download from *VMware Tanzu Network*.

Prerequisites

- The DataDirect JDBC Driver requires Java SE 5 or higher. See *System and Product Requirements* in the DataDirect documentation for information and requirements associated with specific features of the JDBC driver.
- The license key is embedded in the `greenplum.jar` file itself. You do not need to apply a specific license key to the driver to activate it.

Downloading the DataDirect JDBC Driver

To install the JDBC driver on your client:

1. Log into *VMware Tanzu Network* and download the DataDirect JDBC driver file: `PROGRESS_DATADIRECT_JDBC_DRIVER_PIVOTAL_GREENPLUM_5.1.4.zip`.
2. Extract the downloaded ZIP file.
3. Add the full path to the `PROGRESS_DATADIRECT_JDBC_DRIVER_PIVOTAL_GREENPLUM_5.1.4.jar` to your Java `CLASSPATH` environment variable, or add it to your classpath with the `-classpath` option when executing a Java application.

Obtaining Version Details for the Driver

To view the JDBC driver version information:

1. Change to the directory that contains the downloaded `PROGRESS_DATADIRECT_JDBC_DRIVER_PIVOTAL_GREENPLUM_5.1.4.jar` driver file. For example:

```
$ cd /opt/Progress/DataDirect/Connect_for_JDBC_51/lib
```

2. Execute the data source class to display the version information.

   For Linux/Unix systems:

```
$ java -classpath
 PROGRESS_DATADIRECT_JDBC_DRIVER_PIVOTAL_GREENPLUM_5.1.4.jar
 com.pivotal.jdbc.GreenplumDriver
[Pivotal][Greenplum JDBC Driver]Driver Version: 5.1.4.000223
 (F000432.U000208)
```

   For Windows systems:

```
java -classpath .;.\greenplum.jar com.pivotal.jdbc.GreenplumDriver
[Pivotal][Greenplum JDBC Driver]Driver Version: 5.1.4.000223
 (F000432.U000208)
```

Usage Information

The JDBC driver is provided in the `greenplum.jar` file. Use the following data source class and connection URL information with the driver.

Property	Description
Driver File Name	`greenplum.jar`
Data Source Class	com.pivotal.jdbc.GreenplumDriver
Connection URL	jdbc:pivotal:greenplum://host:port;DatabaseName=<name>
Driver Defaults	FetchTWFSasTime=true MaxLongVarcharSize=8190 MaxNumericPrecision=28 MaxNumericScale=6 PrepareThreshold=0 ResultSetMetadataOptions=1 SupportsCatalogs=true

Configuring Prepared Statement Execution

The DataDirect JDBC driver version 5.1.4.000270 (F000450.U000214) introduced support for the `PrepareThreshold` connection property. This property specifies the number of prepared statement executions to be performed before the driver switches to using server-side prepared statements.

The `PrepareThreshold` default value is `0`, always use server-side prepare for prepared statements. This setting preserves the behavior of previous versions of the JDBC driver.

When the `PrepareThreshold` value is greater than 1, it specifies on which execution of a prepared statement the driver starts using server-side prepared statements.

> **Note:** `statement.executeBatch()` always uses server-side prepare for prepared statements. This matches the behavior of the PostgreSQL open source JDBC driver.

Refer to *PrepareThreshold* in the DataDirect documentation for additional information about this connection property.

Limitation

When the `PrepareThreshold` value is greater than one and the prepared statement includes parameterized operations, the driver does not send any SQL prepare calls during `connection.prepareStatement()`. The driver instead sends the query all at once, at execution time. This requires that the driver determine the data types of every column *before* it sends the query to the server. While the driver can make this determination for many data types, it cannot for the JDBC types that can be mapped to multiple Greenplum data types:

- BIT VARYING
- BOOLEAN
- JSON
- TIME WITH TIME ZONE
- UUIDCOL

To work around this limitation, set `PrepareThreshold` to `0` when a prepared statement uses parameterized values with any of the above data types. And use `ResultSet.getMetaData()` to determine if any of the above types are used in a query in advance of submitting the prepared statement.

> **Note:** GPORCA does not support prepared statements that have parameterized values, and will fall back to using the Postgres Planner.

DataDirect Driver Documentation

For more information on working with the Data Direct JDBC driver, see documentation available online at *https://www.progress.com/documentation/datadirect-connectors*. Titles include:

- *User's Guide*
- *Reference*
- *Installation Help*
- *Readme*
- *Quick Start*