DataStax

Apache Cassandra™ Documentation

February 16, 2012

Contents

Apache Cassandra 1.0 Documentation 1

Introduction to Apache Cassandra 1

Getting Started with Cassandra 1

Java Prerequisites 1

Download the Software 1

Install the Software 1

Start the Cassandra Server 1

Login to Cassandra 1

Create a Keyspace (database) 1

Create a Column Family 2

Insert, Update, Delete, Read Data 2

Getting Started with Cassandra and DataStax Community Edition 2

Installing a Single-Node Instance of Cassandra 2

Checking for a Java Installation 2

Installing the DataStax Community Binaries on Linux 3

Configuring and Starting a Single-Node Cluster on Linux 4

Installing the DataStax Community Binaries on Mac 5

Installing the DataStax Community Binaries on Windows 5

Configuring and Starting DataStax OpsCenter 5

Running the Portfolio Demo Sample Application 6

About the Portfolio Demo Use Case 6

Running the Demo Web Application 6

Exploring the Sample Data Model 7

Looking at the Schema Definitions in Cassandra-CLI 8

DataStax Community Release Notes 8

What's New 8

Prerequisites 8

Understanding the Cassandra Architecture 8

About Internode Communications (Gossip) 8

About Cluster Membership and Seed Nodes 9

About Failure Detection and Recovery 9

About Data Partitioning in Cassandra 10

About Partitioning in Multi-Data Center Clusters 10

Understanding the Partitioner Types 12

About the Random Partitioner 12

About Ordered Partitioners 13

About Replication in Cassandra 13

About Replica Placement Strategy 14
SimpleStrategy 14
NetworkTopologyStrategy 14
About Snitches 17
SimpleSnitch 18
DseSimpleSnitch 18
RackInferringSnitch 18
PropertyFileSnitch 19
EC2Snitch 19
EC2MultiRegionSnitch 19
About Dynamic Snitching 19
About Client Requests in Cassandra 19
About Write Requests 20
About Multi-Data Center Write Requests 20
About Read Requests 21
Planning a Cassandra Cluster Deployment 22
Selecting Hardware 22
Memory 22
CPU 22
Disk 23
Network 23
Planning an Amazon EC2 Cluster 23
Capacity Planning 24
Calculating Usable Disk Capacity 24
Calculating User Data Size 24
Choosing Node Configuration Options 25
Storage Settings 25
Gossip Settings 25
Purging Gossip State on a Node 25
Partitioner Settings 25
Snitch Settings 26
Configuring the PropertyFileSnitch 26
Choosing Keyspace Replication Options 27
Installing and Initializing a Cassandra Cluster 27
Installing Cassandra Using the Packaged Releases 27
Creating the Cassandra User and Configuring sudo 27
Installing Cassandra RPM Packages 28
Installing Sun JRE on RedHat Systems 28
Installing Cassandra Debian Packages 29

Installing Sun JRE on Ubuntu Systems 30
About Packaged Installs 31
Next Steps 31
Installing the Cassandra Tarball Distribution 31
About Cassandra Binary Installations 32
Installing JNA 32
Next Steps 32
Initializing a Cassandra Cluster on Amazon EC2 Using the DataStax AMI 32
Creating an EC2 Security Group for DataStax Community Edition 33
Launching the DataStax Community AMI 34
Connecting to Your Cassandra EC2 Instance 35
Configuring and Starting a Cassandra Cluster 38
Initializing a Multi-Node or Multi-Data Center Cluster 38
Calculating Tokens 39
Calculating Tokens for Multiple Racks 40
Calculating Tokens for a Single Data Center 40
Calculating Tokens for a Multi-Data Center Cluster 41
Starting and Stopping a Cassandra Node 42
Starting/Stopping Cassandra as a Stand-Alone Process 42
Starting/Stopping Cassandra as a Service 42
Upgrading Cassandra 43
Best Practices for Upgrading Cassandra 43
Upgrading Cassandra: 0.8.x to 1.0.x 43
New and Changed Parameters between 0.8 and 1.0 44
Upgrading Between Minor Releases of Cassandra 1.0.x 45
Understanding the Cassandra Data Model 45
The Cassandra Data Model 45
Comparing the Cassandra Data Model to a Relational Database 45
About Keyspaces 47
Defining Keyspaces 47
About Column Families 48
About Columns 49
About Special Columns (Counter, Expiring, Super) 49
About Expiring Columns 49
About Counter Columns 50
About Super Columns 50
About Data Types (Comparators and Validators) 50
About Validators 51
About Comparators 51

About Column Family Compression 52
When to Use Compression 52
Configuring Compression on a Column Family 52
About Indexes in Cassandra 52
About Primary Indexes 53
About Secondary Indexes 53
Building and Using Secondary Indexes 53
Planning Your Data Model 54
Start with Queries 54
Denormalize to Optimize 54
Planning for Concurrent Writes 54
Using Natural or Surrogate Row Keys 54
UUID Types for Column Names 55
Managing and Accessing Data in Cassandra 55
About Writes in Cassandra 55
About Compaction 55
About Transactions and Concurrency Control 55
About Inserts and Updates 56
About Deletes 56
About Hinted Handoff Writes 57
About Reads in Cassandra 57
About Data Consistency in Cassandra 58
Tunable Consistency for Client Requests 58
About Write Consistency 58
About Read Consistency 58
Choosing Client Consistency Levels 59
Consistency Levels for Multi-Data Center Clusters 59
Specifying Client Consistency Levels 60
About Cassandra's Built-in Consistency Repair Features 60
Cassandra Client APIs 60
About Cassandra CLI 60
About CQL 61
Other High-Level Clients 61
Java: Hector Client API 61
Python: Pycassa Client API 61
PHP: Phpcassa Client API 61
Getting Started Using the Cassandra CLI 61
Creating a Keyspace 62
Creating a Column Family 62

Creating a Counter Column Family 63

Inserting Rows and Columns 63

Reading Rows and Columns 64

Setting an Expiring Column 64

Indexing a Column 64

Deleting Rows and Columns 65

Dropping Column Families and Keyspaces 65

Getting Started with CQL 65

Starting the CQL Command-Line Program (cqlsh) 65

Running CQL Commands with cqlsh 66

Creating a Keyspace 66

Creating a Column Family 66

Inserting and Retrieving Columns 66

Adding Columns with ALTER COLUMNFAMILY 66

Altering Column Metadata 67

Specifying Column Expiration with TTL 67

Dropping Column Metadata 67

Indexing a Column 67

Deleting Columns and Rows 67

Dropping Column Families and Keyspaces 68

Configuration 68

Node and Cluster Configuration (cassandra.yaml) 68

Node and Cluster Initialization Properties 70

auto_bootstrap 70

broadcast_address 70

cluster_name 70

commitlog_directory 70

data_file_directories 70

initial_token 70

listen_address 70

partitioner 71

rpc_address 71

rpc_port 71

saved_caches_directory 71

seed_provider 71

seeds 71

storage_port 71

endpoint_snitch 71

Performance Tuning Properties 72

- column_index_size_in_kb 72
- commitlog_sync 72
- commitlog_sync_period_in_ms 72
- commitlog_total_space_in_mb 72
- compaction_preheat_key_cache 72
- compaction_throughput_mb_per_sec 72
- concurrent_compactors 72
- concurrent_reads 72
- concurrent_writes 72
- flush_largest_memtables_at 73
- in_memory_compaction_limit_in_mb 73
- index_interval 73
- memtable_flush_queue_size 73
- memtable_flush_writers 73
- memtable_total_space_in_mb 73
- multithreaded_compaction 73
- reduce_cache_capacity_to 73
- reduce_cache_sizes_at 73
- sliced_buffer_size_in_kb 74
- stream_throughput_outbound_megabits_per_sec 74

Remote Procedure Call Tuning Properties 74

- request_scheduler 74
- request_scheduler_id 74
- request_scheduler_options 74
  - throttle_limit 74
  - default_weight 74
  - weights 74
- rpc_keepalive 74
- rpc_max_threads 75
- rpc_min_threads 75
- rpc_recv_buff_size_in_bytes 75
- rpc_send_buff_size_in_bytes 75
- rpc_timeout_in_ms 75
- rpc_server_type 75
- thrift_framed_transport_size_in_mb 75
- thrift_max_message_length_in_mb 75

Internode Communication and Fault Detection Properties 75

- dynamic_snitch 75
- dynamic_snitch_badness_threshold 75

dynamic_snitch_reset_interval_in_ms 76

dynamic_snitch_update_interval_in_ms 76

hinted_handoff_enabled 76

hinted_handoff_throttle_delay_in_ms 76

max_hint_window_in_ms 76

phi_convict_threshold 76

Automatic Backup Properties 76

incremental_backups 76

snapshot_before_compaction 76

Security Properties 76

authenticator 76

authority 77

internode_encryption 77

keystore 77

keystore_password 77

truststore 77

truststore_password 77

Keyspace and Column Family Storage Configuration 77

Keyspace Attributes 78

name 78

placement_strategy 78

strategy_options 78

Column Family Attributes 79

column_metadata 79

column_type 80

comment 80

compaction_strategy 80

compaction_strategy_options 80

comparator 81

compare_subcolumns_with 81

compression_options 81

default_validation_class 81

gc_grace_seconds 81

key_cache_save_period_in_seconds 81

keys_cached 82

key_validation_class 82

name 82

read_repair_chance 82

replicate_on_write 82

max_compaction_threshold 82

min_compaction_threshold 82

memtable_flush_after_mins 82

memtable_operations_in_millions 82

memtable_throughput_in_mb 83

rows_cached 83

row_cache_provider 83

row_cache_save_period_in_seconds 83

Java and System Environment Settings Configuration 83

Heap Sizing Options 83

JMX Options 83

Further Reading on JVM Tuning 84

Authentication and Authorization Configuration 84

access.properties 84

passwd.properties 85

Logging Configuration 85

Logging Levels via the Properties File 85

Logging Levels via JMX 85

Operations 86

Monitoring a Cassandra Cluster 86

Monitoring Using DataStax OpsCenter 86

Monitoring Using `nodetool` 87

Monitoring Using JConsole 88

Compaction Metrics 89

Thread Pool Statistics 90

Read/Write Latency Metrics 90

ColumnFamily Statistics 90

Monitoring and Adjusting Cache Performance 91

Tuning Cassandra 91

Tuning the Cache 92

How Caching Works 92

Configuring the Column Family Key Cache 92

Configuring the Column Family Row Cache 92

Data Modeling Considerations for Cache Tuning 93

Hardware and OS Considerations for Cache Tuning 93

Estimating Cache Sizes 93

Tuning Write Performance (Memtables) 93

Tuning Java Heap Size 93

Tuning Java Garbage Collection 94

Tuning Compaction 94

Choosing a Column Family Compaction Strategy 94

Setting the Compaction Strategy on a Column Family 94

Tuning Options for Size-Tiered Compaction 95

Managing a Cassandra Cluster 95

Running Routine Node Repair 95

Adding Capacity to an Existing Cluster 95

Calculating Tokens For the New Nodes 96

Adding Nodes to a Cluster 96

Changing the Replication Factor 97

Replacing a Dead Node 97

Backing Up and Restoring Data 98

Taking a Snapshot 98

Clearing Snapshot Files 98

Enabling Incremental Backups 98

Restoring from a Snapshot 99

References 99

CQL Language Reference 99

CQL Lexical Structure 99

CQL Identifiers and Keywords 100

CQL Constants 100

CQL Comments 100

CQL Consistency Levels 100

CQL Data Types 101

Working with Dates and Times 101

CQL Storage Parameters 102

CQL Keyspace Storage Parameters 102

CQL Column Family Storage Parameters 102

CQL Commands 102

ALTER COLUMNFAMILY 102

Synopsis 103

Description 103

Parameters 103

Examples 103

BATCH 103

Synopsis 103

Description 104

Parameters 104

Example 104

CREATE COLUMNFAMILY 105
- Synopsis 105
- Description 105
- Parameters 105
- Examples 106

CREATE INDEX 106
- Synopsis 106
- Description 106
- Parameters 106
- Examples 106

CREATE KEYSPACE 107
- Synopsis 107
- Description 107
- Parameters 107
- Examples 107

DELETE 107
- Synopsis 108
- Description 108
- Parameters 108
- Example 108

DROP COLUMNFAMILY 109
- Synopsis 109
- Description 109
- Parameters 109
- Example 109

DROP INDEX 109
- Synopsis 109
- Description 109
- Parameters 109
- Example 109

DROP KEYSPACE 109
- Synopsis 110
- Description 110
- Parameters 110
- Example 110

INSERT 110
- Synopsis 110
- Description 110
- Parameters 110

Example 111

SELECT 111

Synopsis 111

Description 111

Parameters 112

Examples 112

TRUNCATE 113

Synopsis 113

Description 113

Parameters 113

Example 113

UPDATE 113

Synopsis 113

Description 113

Parameters 114

<column_family> 114

Example 114

USE 115

Synopsis 115

Description 115

Parameters 115

Example 115

CQLSH-Specific Commands 115

ASSUME 115

Synopsis 115

Description 116

Parameters 116

Examples 116

DESCRIBE 116

Synopsis 116

Description 116

Parameters 116

Examples 117

SHOW 117

Synopsis 117

Description 117

Parameters 117

Examples 117

nodetool 118

- cassandra 120
    - Usage 120
    - Environment 120
    - Options 120
    - Examples 121
- stress 121
    - Setting up the Stress Utility 122
    - Usage 122
    - Using the Daemon Mode (`stressd`) 123
    - Examples 123
- sstable2json / json2sstable 123
    - `sstable2json` 124
        - Usage 124
        - Output Format 124
    - `json2sstable` 125
        - Usage 125
    - `sstablekeys` 125
        - Usage 125

Troubleshooting Guide 125

- Reads are getting slower while writes are still fast 125
- Nodes seem to freeze after some period of time 126
- Nodes are dying with OOM errors 126
- Nodetool or JMX Connections Failing on Remote Nodes 126
- View of ring differs between some nodes 126
- Java reports an error saying there are too many open files 126

Apache Cassandra 1.0 Documentation

Introduction to Apache Cassandra

Apache Cassandra is a free, open-source, distributed database system for managing large amounts of structured, semi-structured, and unstructured data. Cassandra is designed to scale to a very large size across many commodity servers with no single point of failure. Cassandra provides a powerful dynamic schema data model designed to allow for maximum flexibility and performance at scale.

Getting Started with Cassandra

Getting started with Cassandra is fast and easy. Installing Cassandra on a single machine is the best way to learn the basics. The following will help you install Cassandra and become familiar with some basic commands.

Java Prerequisites

Before installing Cassandra on Linux, Windows, or Mac, ensure that you have the most up-to-date version of Java installed on your machine. To determine if Java is installed on your system, in a terminal window enter:

```
java -version
```

Download the Software

Download the DataStax Community Edition Server, which is a bundle containing the most up-to-date version of Cassandra along with all the utilities and tools you'll need. You can also download directly from a terminal window `wget` on Linux or `curl` on Mac and the following URL:

```
http://downloads.datastax.com/community/dsc.tar.gz
```

On Windows, download and use the DataStax Windows MSI installer.

Note that DataStax also makes available RPM and Debian builds for Linux that are available on the main Download page.

Install the Software

For Linux and Mac machines, unzip the tar download file:

```
tar -xvf <tar file>
```

Start the Cassandra Server

On Linux or Mac, navigate to the `bin` directory and invoke the cassandra script:

```
sudo ./cassandra
```

On Windows, the Cassandra server starts running after installation.

Login to Cassandra

Cassandra has a couple of interfaces that you can use to enter commands - the CLI and the CQL (Cassandra Query Language) utility. For this indroduction use the CLI, which in the Windows Cassandra group folder and in the bin directory on Linux or Mac:

```
./cassandra-cli -h localhost
```

Create a Keyspace (database)

A keyspace in Cassandra is the equivalent of a database in the RDBMS world. You can have multiple keyspaces on a Cassandra server.

First create a simple keyspace to work with:

```
[default@unknown] create keyspace mykeyspace;
```

```
[default@unknown] use mykeyspace;
```

Create a Column Family

A couple family is the primary data object in Cassandra, similar to a table in the RDBMS world. To create simple column family:

```
[default@mykeyspace] create column family cf1;
```

Insert, Update, Delete, Read Data

Now you can enter and read data from Cassandra.

To insert data:

```
[default@mykeyspace] set cf1[1]['c2']=utf8('test');
```

To read that data:

```
[default@mykeyspace] get cf1[1];
```

To update that data:

```
[default@mykeyspace] set cf1[1]['c2']=utf8('test2');
```

To delete that data:

```
[default@mykeyspace] del cf1[1];
```

To exit the CLI:

```
[default@mykeyspace] exit;
```

Getting Started with Cassandra and DataStax Community Edition

This quick start guide is intended to get you up and running quickly on a single-node instance using the DataStax Community Edition packages. DataStax Community Edition is a smart bundle comprised of the most up-to-date and stable version of Apache Cassandra, DataStax OpsCenter Community Edition, the CQL command line utility, and the DataStax portfolio demo application.

This tutorial will guide you through setting up a single-node cluster in your home directory, and running the demo application to see Cassandra in action.

Installing a Single-Node Instance of Cassandra

The fastest way to get up and running quickly with Cassandra is to install Cassandra using the DataStax Community tarball distributions and start a single-node instance. Cassandra is intended to be run on multiple nodes, however starting out with a single-node cluster is a great way to get started.

Getting up and running takes just three simple steps:

1. *Make sure you have Java installed*
2. *Install the DataStax Community Edition of Apache Cassandra*
3. *Set a couple of configuration properties and start the Cassandra server*

Checking for a Java Installation

Cassandra is a Java program and requires a Java Virtual Machine (JVM) to be installed before you can start the server. For production deployments, you will need the Sun Java Runtime Environment 1.6.0_19 or later, but if you are just installing an evaluation instance, any JVM is fine.

To check for Java, run the following command in a terminal session:

```
# java -version
```

If you do not have Java installed, see *Installing Sun JRE on RedHat Systems* or *Installing Sun JRE on Ubuntu Systems* for instructions.

Installing the DataStax Community Binaries on Linux

The quickest way to get going on a single node with Cassandra is to install the DataStax Community Edition binary tarball packages. This allows you to install everything in a single location (such as your home directory), and does not require root permissions.

DataStax Community is comprised of three components - The Apache Cassandra server, the DataStax portfolio demo application, and DataStax OpsCenter (a web-based monitoring application for Cassandra).

Note

The instructions in this section are not intended for production installations, just for a quick start tutorial. See *Planning a Cassandra Cluster Deployment*, *Installing and Initializing a Cassandra Cluster*, and *Configuring and Starting a Cassandra Cluster* for production cluster setup best practices.

These instructions will walk you through setting up a self-contained, single-node instance of Cassandra in your home directory (does not require root permissions).

Note

By downloading community software from DataStax you agree to the terms of the DataStax Community EULA (End User License Agreement) posted on the DataStax web site.

1. In your home directory, create a directory called `datastax`.

   ```
   $ cd $HOME

   $ mkdir datastax

   $ cd datastax
   ```

2. In the `datastax` directory download the Cassandra package (required), plus the OpsCenter package (optional).

   For example, on Linux to get version 1.0.7 of DataStax Community and version 1.4 of OpsCenter:

   ```
   $ wget http://downloads.datastax.com/community/dsc.tar.gz

   $ wget http://downloads.datastax.com/community/opscenter.tar.gz

   $ wget http://downloads.datastax.com/community/dsc-1.0.1-demo-bin.tar.gz
   ```

3. Unpack the distributions:

```
$ tar -xzvf dsc.tar.gz

$ tar -xzvf opscenter.tar.gz

$ tar -xzvf dsc-1.0.1-demo-bin.tar.gz

$ rm *.tar.gz
```

4. For convenience, set the following environment variables in your user environment. For example, to configure your environment in your `$HOME/.bashrc` file:

   a. Open your .bashrc file in a text editor (such as `vi`):

   ```
   vi $HOME/.bashrc
   ```

   b. Add the following lines to bottom of the file:

   ```
   export CASSANDRA_HOME=$HOME/datastax/<dsc_package_name>
   export DSCDEMO_HOME=$HOME/datastax/dsc-1.0.1/demos/portfolio_manager
   export OPSC_HOME=$HOME/datastax/<opscenter_package_name>
   export PATH="$PATH:$CASSANDRA_HOME/bin:$DSCDEMO_HOME/bin:$OPSC_HOME/bin"
   ```

   For example: `<dsc_package_name>` = `dsc-cassandra-1.0.7` and `<opscenter_package_name>` = `opscenter-1.4`

   c. Save and close the file.

   d. Source the file.

   ```
   source $HOME/.bashrc
   ```

5. Create the data and logging directory for Cassandra.

```
$ mkdir $HOME/datastax/cassandra-data
```

Configuring and Starting a Single-Node Cluster on Linux

1. Set the configuration properties needed to start your cluster in the `$CASSANDRA_HOME/conf/cassandra.yaml` file. This will configure Cassandra to run a single-node cluster on the localhost and store all of its data files in your home directory.

```
$ sed -i -e "s,initial_token:,initial_token: 0," \
  $CASSANDRA_HOME/conf/cassandra.yaml

$ sed -i -e "s,- /var/lib/cassandra/data,- $HOME/datastax/cassandra-data," \
  $CASSANDRA_HOME/conf/cassandra.yaml

$ sed -i -e "s,saved_caches_directory: /var/lib/cassandra/saved_caches, \
  saved_caches_directory: $HOME/datastax/cassandra-data/saved_caches," \
  $CASSANDRA_HOME/conf/cassandra.yaml

$ sed -i -e "s,commitlog_directory: /var/lib/cassandra/commitlog,commitlog_directory: \
  $HOME/datastax/cassandra-data/commitlog," $CASSANDRA_HOME/conf/cassandra.yaml
```

2. Set the Cassandra server log location in the `$CASSANDRA_HOME/conf/log4j-server.properties` file to the log to the `cassandra-data` directory you created earlier:

   ```
   $ sed -i -e "s,log4j.appender.R.File=/var/log/cassandra/system.log, \
     log4j.appender.R.File=$HOME/datastax/cassandra-data/system.log," \
     $CASSANDRA_HOME/conf/log4j-server.properties
   ```

3. Configure the DataStax demo application to point to the correct Cassandra installation location:

   ```
   $ sed -i -e "s,/usr/share/cassandra,$HOME/datastax/<dsc_package_name>," \
     $DSCDEMO_HOME/bin/pricer
   ```

4. Start the Cassandra server in the background.

   ```
   $ cassandra
   ```

5. Check that your Cassandra ring is up and running:

   ```
   $ nodetool ring -h localhost
   ```

6. For the next step, run the *Portfolio Demo* example application.

Installing the DataStax Community Binaries on Mac

DataStax supplies a tar download for Mac that can be used for development purposes (production deployments on Mac are not currently supported). To install the DataStax Community server and sample applications on Mac, follow these instructions:

1. Download the tar package for Mac from the DataStax website.
2. Move the tar package to your target directory and unpack the contents. An example of unpacking version 1.0.7 of DataStax Community would be: `tar –xzvf dsc-cassandra-1.0.7-bin.tar.gz`.
3. If you want to take the default data and log file locations for Cassandra, you can proceed to starting up Cassandra by going to the bin directory of the installation home directory and entering the command `sudo ./cassandra`. If you want to configure Cassandra to use other directories for its data and log files, you can modify the cassandra.yaml file (located in the `/conf` directory) and change the `data_file_directories`, `commitlog_directory`, and `saved_caches_directory` parameters to point to your desired directory locations.
4. Check that Cassandra is running by invoking the nodetool utility from the installation home directory: `./bin/nodetool ring –h localhost`.

Installing the DataStax Community Binaries on Windows

DataStax provides a GUI installer for installing both OpsCenter and Cassandra on Windows. Simply download the Windows installer for your chosen platform (32- or 64-bit Windows 7 or Windows Server 2008) from the DataStax website and follow the installation wizard to install Cassandra, the sample applications, and OpsCenter.

Note

There is a dependency on the Visual C++ 2008 runtime (32-bit). However, Windows 7 and Windows 2008 Server R2 already have it installed. If needed, you can download it from http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=29.

Configuring and Starting DataStax OpsCenter

DataStax OpsCenter is a graphical web application that can be used to manage and monitor a Cassandra cluster. For specific information on installing OpsCenter for your chosen platform, see the Installing the OpsCenter Dashboard section of the online OpsCenter Documentation.

Running the Portfolio Demo Sample Application

Your DataStax Community (DSC) installation contains a demo portfolio manager application that showcases how you can use Apache Cassandra to back a real-time web application developed in Java and using the Cassandra Query Language (CQL) JDBC driver.

The demo application is located in:

- RPM and Debian packaged installations: `/usr/share/dse-demos/portfolio_manager`
- Binary tar file installations: `$DSCDEMO_HOME` (*<install_location>/dsc-1.0.1/demos/portfolio_manager`*).

About the Portfolio Demo Use Case

The portfolio manager sample application is a financial application where users can actively create and manage a portfolio of stocks. Each portfolio contains a list of stocks, the number of shares purchased, and the price at which the shares were purchased. An overall value is maintained for each stock portfolio as well as the percentage of gain or loss compared to the original stock purchase prices for a portfolio.

The application has a `pricer` utility which is meant to simulate an active feed of live stock market data. For each stock ticker symbol, the application tracks the current stock price and historical market data (end-of-day price) for each stock going back in time.

Running the Demo Web Application

Before you begin, make sure you have installed, configured, and started your Cassandra cluster. Also make sure you have installed the DataStax Community Edition demo package using either the binary tarball distribution or the RPM/Debian packaged installations (see *Installing a Single-Node Instance of Cassandra* or *Installing Cassandra Using the Packaged Releases*).

1. Go to the portfolio manager demo directory.

   ```
   cd $DSCDEMO_HOME
   ```

   or in packaged installs:

   ```
   cd /usr/share/dse-demos/portfolio_manager
   ```

   ### Note

   You must run the `pricer` utility from a directory where you have write permissions (such as your home directory), or else run it as root or using sudo.

2. Run the `./bin/pricer` utility to generate stock data for the application. To see all of the available options for this utility:

   ```
   ./bin/pricer --help
   ```

   The following examples will generate 100 days worth of historical data.

   If running on a single node cluster on localhost:

   ```
   ./bin/pricer -o INSERT_PRICES
   ./bin/pricer -o UPDATE_PORTFOLIOS
   ./bin/pricer -o INSERT_HISTORICAL_PRICES -n 100
   ```

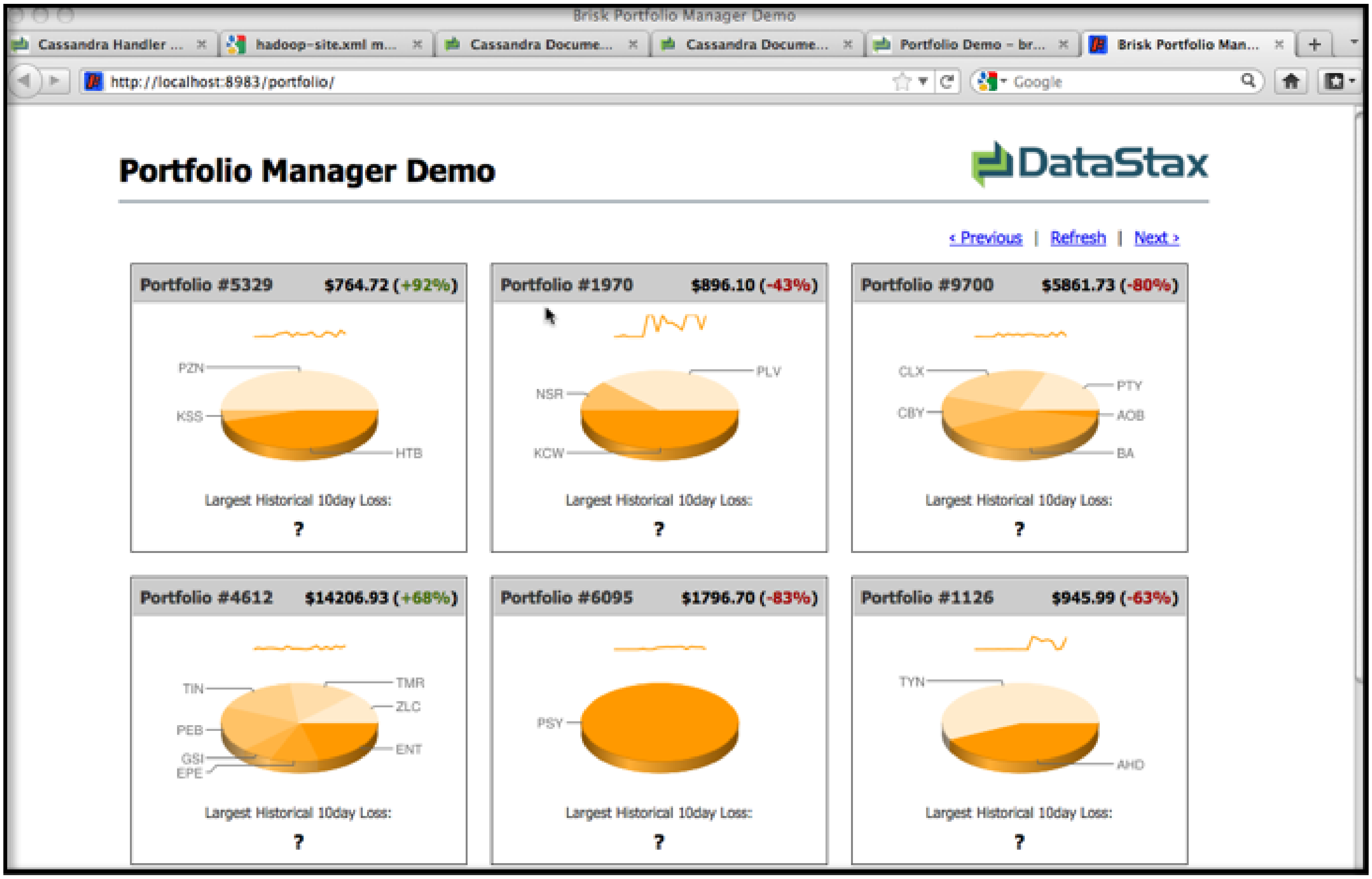
4. Start the web service (must be in the `$DSCDEMO_HOME/website` directory to start).

```
$ cd $DSCDEMO_HOME/website

$ java -jar start.jar &
```

5. Open a browser and go to `http://localhost:8983/portfolio` (if running on the local machine) or `http://<webhost_ip>:8983/portfolio` (if running remotely - specify the correct IP address of the remote server).

This will open the Portfolio Manager demo web application home page.


Brisk Portfolio Manager Demo
Cassandra Handler ...
hadoop-site.xml m...
Cassandra Docume...
Cassandra Docume...
Portfolio Demo - br...
Brisk Portfolio Man...
http://localhost:8983/portfolio/
Google
Portfolio Manager Demo
DataStax
‹ Previous | Refresh | Next ›
Portfolio #5329 $764.72 (+92%)
PZN
KSS
HTB
Largest Historical 10day Loss:
?
Portfolio #1970 $896.10 (-43%)
PLV
NSR
KCW
Largest Historical 10day Loss:
?
Portfolio #9700 $5861.73 (-80%)
CLX
PTY
CBY
AOB
BA
Largest Historical 10day Loss:
?
Portfolio #4612 $14206.93 (+68%)
TIN
TMR
ZLC
PEB
ENT
GSI
EPE
Largest Historical 10day Loss:
?
Portfolio #6095 $1796.70 (-83%)
PSY
Largest Historical 10day Loss:
?
Portfolio #1126 $945.99 (-63%)
TYN
AHD
Largest Historical 10day Loss:
?

Exploring the Sample Data Model

The data for the portfolio manager sample application is contained in a Cassandra keyspace called `PortfolioDemo`.

In that keyspace are four column families:

- **Portfolio** - One row per portfolio/customer where the column names are the stock ticker symbols and the column values are the current stock price.
- **StockHist** - One row per stock ticker symbol with (time-ordered) dates for the column names and column values are the end-of-day price for a particular day.
- **Stocks** - One row per stock ticker symbol with a static column name *price* and the column value is the current stock value.
- **HistLoss** - One row per stock ticker symbol where the column name is the worst date in the stock's history in the form of YYYY-MM-DD and the column value is the loss dollar amount.

Looking at the Schema Definitions in Cassandra-CLI

The `cassandra-cli` program in a command-line interface for Cassandra. Using `cassandra-cli` you can explore the PortfolioDemo keyspace and data model.

1. Start `cassandra-cli` and specify a Cassandra node to connect to. For example, if running a single-node instance on `localhost`:

   ```
   $ cassandra-cli -h localhost
   ```

2. Specify the keyspace you want to connect to:

   ```
   [default@unknown] USE PortfolioDemo;
   ```

3. To see the keyspace and column family schema definitions:

   ```
   [default@unknown] SHOW SCHEMA;
   ```

4. To select a row from the `Stocks` column family (by specifying the row key value of a stock ticker symbol):

   ```
   [default@unknown] GET Stocks[GCO];
   ```

5. To exit `cassandra-cli`:

   ```
   [default@unknown] exit;
   ```

DataStax Community Release Notes

What's New

Added new platform support (Windows 7 and Windows 2008 Server, both 32- and 64-bit) for Cassandra development. The Windows MSI installer provides a full install, including OpsCenter, sets all the WIN services, creates a Windows program group, and quickly starts the new version of DataStax OpsCenter (1.4).

Prerequisites

Dependency on the Visual C++ 2008 runtime (32-bit). However, Windows 7 and Windows 2008 Server R2 already have it installed. If needed, download it from http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=29.

If you just want to learn more about Cassandra and how it works, see the following conceptual topics:

- *Understanding the Cassandra Architecture*
- *Understanding the Cassandra Data Model*
- *Managing and Accessing Data in Cassandra*

Understanding the Cassandra Architecture

A Cassandra instance is a collection of independent nodes that are configured together into a *cluster*. In a Cassandra cluster, all nodes are peers, meaning there is no master node or centralized management process. A node joins a Cassandra cluster based on certain aspects of its configuration. This section explains those aspects of the Cassandra cluster architecture.

About Internode Communications (Gossip)

Cassandra uses a protocol called *gossip* to discover location and state information about the other nodes participating in a Cassandra cluster. Gossip is a peer-to-peer communication protocol in which nodes periodically exchange state information about themselves and about other nodes they know about.

In Cassandra, the gossip process runs every second and exchanges state messages with up to three other nodes in the cluster. The nodes exchange information about themselves and about the other nodes that they have gossiped about, so all nodes quickly learn about all other nodes in the cluster. A gossip message has a version associated with it, so that during a gossip exchange, older information is overwritten with the most current state for a particular node.

About Cluster Membership and Seed Nodes

When a node first starts up, it looks at its configuration file to determine the name of the Cassandra cluster it belongs to and which node(s), called *seeds*, to contact to obtain information about the other nodes in the cluster. These cluster contact points are configured in the *cassandra.yaml* configuration file for a node.

To prevent partitions in gossip communications, all nodes in a cluster should have the *same* list of seed nodes listed in their configuration file. This is most critical the *first* time a node starts up. By default, a node will remember other nodes it has gossiped with between subsequent restarts.

Note

The seed node designation has no purpose other than bootstrapping the gossip process for new nodes joining the cluster. Seed nodes are *not* a single point of failure, nor do they have any other special purpose in cluster operations beyond the bootstrapping of nodes.

To know what range of data it is responsible for, a node must also know its own *token* and those of the other nodes in the cluster. When initializing a new cluster, you should generate tokens for the entire cluster and assign an initial token to each node before starting up. Each node will then gossip its token to the others. See *About Data Partitioning in Cassandra* for more information about partitioners and tokens.

About Failure Detection and Recovery

Failure detection is a method for locally determining, from gossip state, if another node in the system is up or down. Failure detection information is also used by Cassandra to avoid routing client requests to unreachable nodes whenever possible. (Cassandra can also avoid routing requests to nodes that are alive, but performing poorly, through the *dynamic snitch*.)

The gossip process tracks heartbeats from other nodes both directly (nodes gossiping directly to it) and indirectly (nodes heard about secondhand, thirdhand, and so on). Rather than have a fixed threshold for marking nodes without a heartbeat as down, Cassandra uses an accrual detection mechanism to calculate a per-node threshold that takes into account network conditions, workload, or other conditions that might affect perceived heartbeat rate. During gossip exchanges, every node maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster. The value of phi is based on the distribution of inter-arrival time values across all nodes in the cluster. In Cassandra, configuring the *phi_convict_threshold* property adjusts the sensitivity of the failure detector. The default value is fine for most situations, but DataStax recommends increasing it to `12` for Amazon EC2 due to the network congestion frequently experienced on that platform.

Node failures can result from various causes such as hardware failures, network outages, and so on. Node outages are often transient but can last for extended intervals. A node outage rarely signifies a permanent departure from the cluster, and therefore does not automatically result in permanent removal of the node from the ring. Other nodes will still try to periodically initiate gossip contact with failed nodes to see if they are back up. To permanently change a node's membership in a cluster, administrators must explicitly add or remove nodes from a Cassandra cluster using the *nodetool* utility.

When a node comes back online after an outage, it may have missed writes for the replica data it maintains. Once the failure detector marks a node as down, missed writes are stored by other replicas if *hinted handoff* is enabled (for a period of time, anyways). However, it is possible that some writes were missed between the interval of a node actually going down and when it is detected as down. Or if a node is down for longer than *max_hint_window_in_ms* (one hour by default), hints will no longer be saved. For that reason, it is best practice to routinely run *nodetool repair* on all nodes to ensure they have consistent data, and to also run repair after recovering a node that has been down for an extended period.

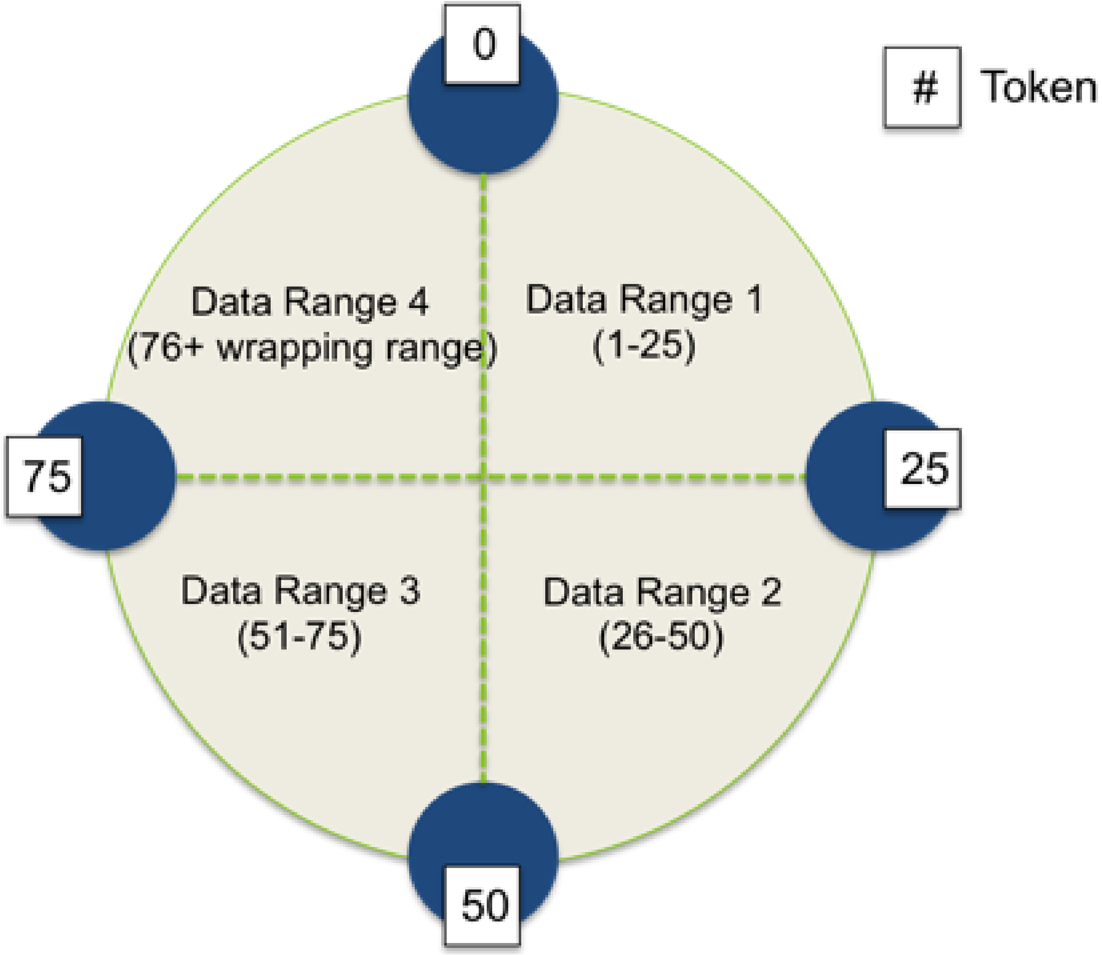
About Data Partitioning in Cassandra

When you start a Cassandra cluster, you must choose how the data will be divided across the nodes in the cluster. This is done by choosing a *partitioner* for the cluster.

In Cassandra, the total data managed by the cluster is represented as a circular space or *ring*. The ring is divided up into ranges equal to the number of nodes, with each node being responsible for one or more ranges of the overall data. Before a node can join the ring, it must be assigned a token. The token determines the node's position on the ring and the range of data it is responsible for.

Column family data is partitioned across the nodes based on the row key. To determine the node where the first replica of a row will live, the ring is walked clockwise until it locates the node with a token value greater than that of the row key. Each node is responsible for the region of the ring between itself (inclusive) and its predecessor (exclusive). With the nodes sorted in token order, the last node is considered the predecessor of the first node; hence the ring representation.

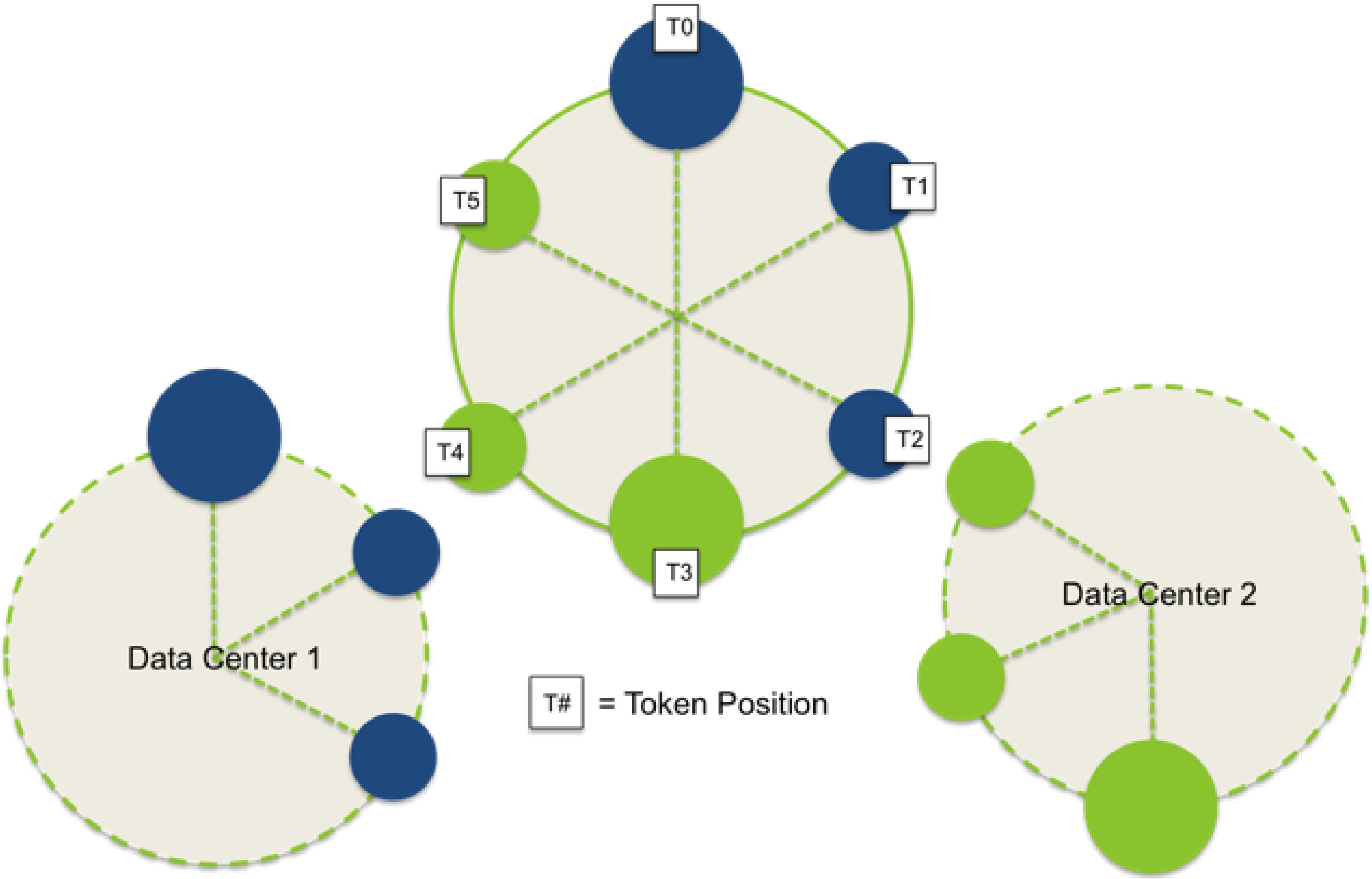
For example, consider a simple 4 node cluster where all of the row keys managed by the cluster were numbers in the range of 0 to 100. Each node is assigned a token that represents a point in this range. In this simple example, the token values are 0, 25, 50, and 75. The first node, the one with token 0, is responsible for the *wrapping range* (75-0). The node with the lowest token also accepts row keys less than the lowest token and more than the highest token.



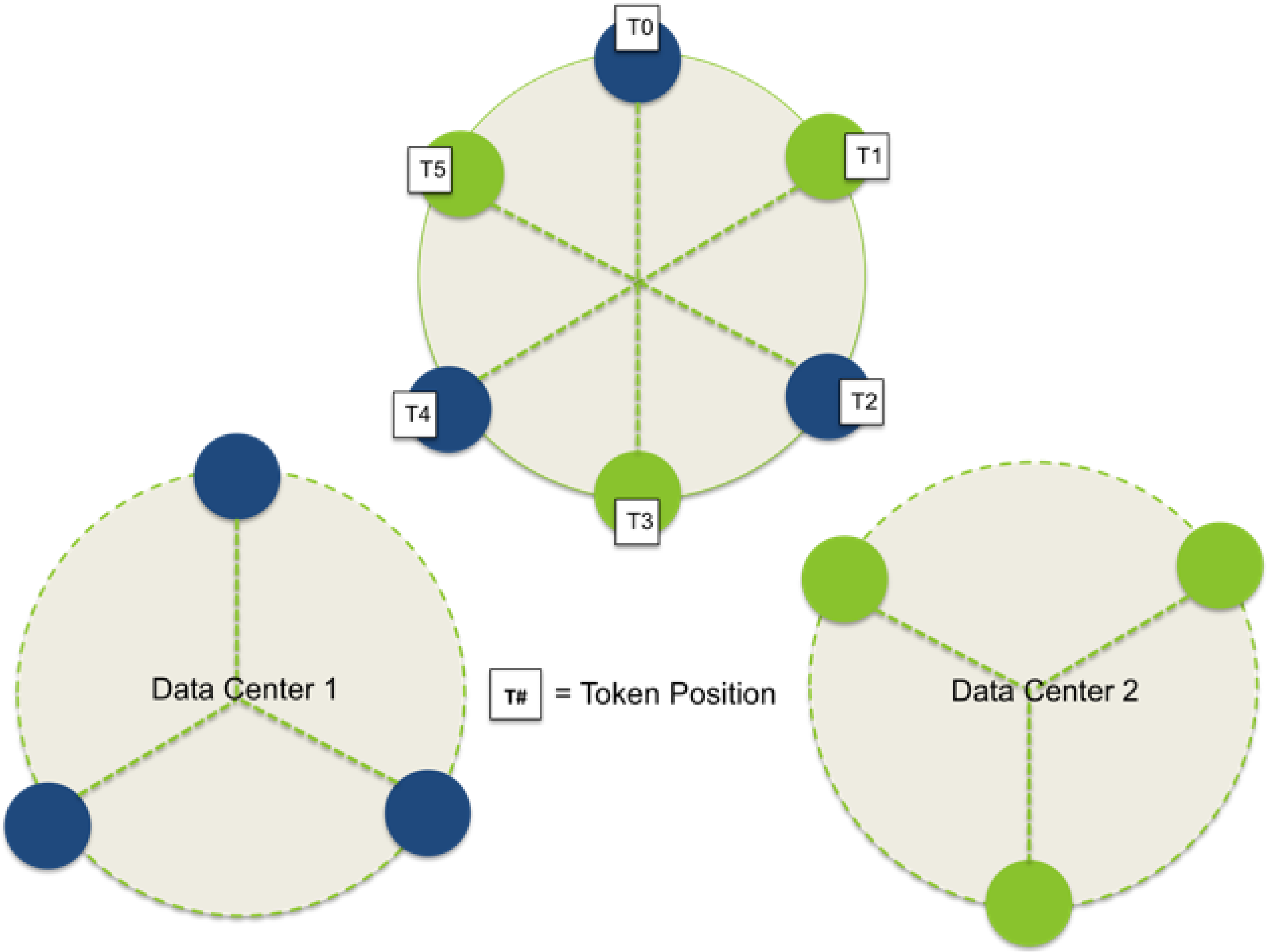
About Partitioning in Multi-Data Center Clusters

In multi-data center deployments, replica placement is calculated per data center when using the `NetworkTopologyStrategy` replica placement strategy. In each data center (or replication group) the first replica for a particular row is determined by the token value assigned to a node. Additional replicas in the same data center are placed by walking the ring clockwise until it reaches the first node in another rack.

If you do not calculate partitioner tokens so that the data ranges are evenly distributed for each data center, you could end up with uneven data distribution within a data center.



The goal is to ensure that the nodes for each data center have token assignments that evenly divide the overall range. Otherwise, you could end up with nodes in each data center that own a disproportionate number of row keys. Each data center should be partitioned as if it were its own distinct ring, however token assignments within the entire cluster cannot conflict with each other (each node must have a unique token). See *Calculating Tokens for a Multi-Data Center Cluster* for strategies on how to generate tokens for multi-data center clusters.



Understanding the Partitioner Types

Unlike almost every other configuration choice in Cassandra, the partitioner may not be changed without reloading all of your data. It is important to choose and configure the correct partitioner before initializing your cluster.

Cassandra offers a number of partitioners out-of-the-box, but the random partitioner is the best choice for most Cassandra deployments.

About the Random Partitioner

The `RandomPartitioner` is the default partitioning strategy for a Cassandra cluster, and in almost all cases is the right choice.

Random partitioning uses *consistent hashing* to determine which node will store a particular row. Unlike naive modulus-by-node-count, consistent hashing ensures that when nodes are added to the cluster, the minimum possible set of data is affected.

To distribute the data evenly across the number of nodes, a hashing algorithm creates an MD5 hash value of the row key. The possible range of hash values is from 0 to 2**127. Each node in the cluster is assigned a *token* that represents a hash value within this range. A node then owns the rows with a hash value less than its token number. For single data center deployments, tokens are calculated by dividing the hash range by the number of nodes in the cluster. For multi data center deployments, tokens are calculated per data center (the hash range should be evenly divided for the nodes in each replication group).

The primary benefit of this approach is that once your tokens are set appropriately, data from all of your column families is evenly distributed across the cluster with no further effort. For example, one column family could be using user names as the row key and another column family timestamps, but the row keys from each individual column family are still spread evenly. This also means that read and write requests to the cluster will also be evenly distributed.

Another benefit of using random partitioning is the simplification of load balancing a cluster. Because each part of the hash range will receive an equal number of rows on average, it is easier to correctly assign tokens to new nodes.

About Ordered Partitioners

Using an ordered partitioner ensures that row keys are stored in sorted order. Unless absolutely required by your application, DataStax strongly recommends choosing the random partitioner over an ordered partitioner.

Using an ordered partitioner allows range scans over rows, meaning you can scan rows as though you were moving a cursor through a traditional index. For example, if your application has user names as the row key, you can scan rows for users whose names fall between Jake and Joe. This type of query would not be possible with randomly partitioned row keys, since the keys are stored in the order of their MD5 hash (not sequentially).

Although having the ability to do range scans on rows sounds like a desirable feature of ordered partitioners, there are ways to achieve the same functionality using column family indexes. Most applications can be designed with a data model that supports ordered queries as slices over a set of columns rather than range scans over a set of rows.

Using an ordered partitioner is not recommended for the following reasons:

- **Sequential writes can cause hot spots.** If your application tends to write or update a sequential block of rows at a time, then the writes will not be distributed across the cluster; they will all go to one node. This is frequently a problem for applications dealing with timestamped data.
- **More administrative overhead to load balance the cluster.** An ordered partitioner requires administrators to manually calculate token ranges based on their estimates of the row key distribution. In practice, this requires actively moving node tokens around to accommodate the actual distribution of data once it is loaded.
- **Uneven load balancing for multiple column families.** If your application has multiple column families, chances are that those column families have different row keys and different distributions of data. An ordered partitioner than is balanced for one column family may cause hot spots and uneven distribution for another column family in the same cluster.

There are three choices of built-in ordered partitioners that come with Cassandra. Note that the `OrderPreservingPartitioner` and `CollatingOrderPreservingPartitioner` are deprecated as of Cassandra 0.7 in favor of the `ByteOrderedPartitioner`:

- **ByteOrderedPartitioner** - Row keys are stored in order of their raw bytes rather than converting them to encoded strings. Tokens are calculated by looking at the actual values of your row key data and using a hexadecimal representation of the leading character(s) in a key. For example, if you wanted to partition rows alphabetically, you could assign an `A` token using its hexadecimal representation of `41`.
- **OrderPreservingPartitioner** - Row keys are stored in order based on the UTF-8 encoded value of the row keys. Requires row keys to be UTF-8 encoded strings.
- **CollatingOrderPreservingPartitioner** - Row keys are stored in order based on the United States English locale (EN_US). Also requires row keys to be UTF-8 encoded strings.

About Replication in Cassandra

Replication is the process of storing copies of data on multiple nodes to ensure reliability and fault tolerance. When you create a keyspace in Cassandra, you must decide the *replica placement strategy*: the number of replicas and how those replicas are distributed across nodes in the cluster. The replication strategy relies on the cluster-configured *snitch* to help it determine the physical location of nodes and their proximity to each other.

The total number of replicas across the cluster is often referred to as the *replication factor*. A replication factor of 1 means that there is only one copy of each row. A replication factor of 2 means two copies of each row. All replicas are equally important; there is no *primary* or *master* replica in terms of how read and write requests are handled.

As a general rule, the replication factor should not exceed the number of nodes in the cluster. However, it is possible to increase replication factor, and then add the desired number of nodes afterwards. When replication factor exceeds the number of nodes, writes will be rejected, but reads will be served as long as the desired *consistency level* can be met.

About Replica Placement Strategy

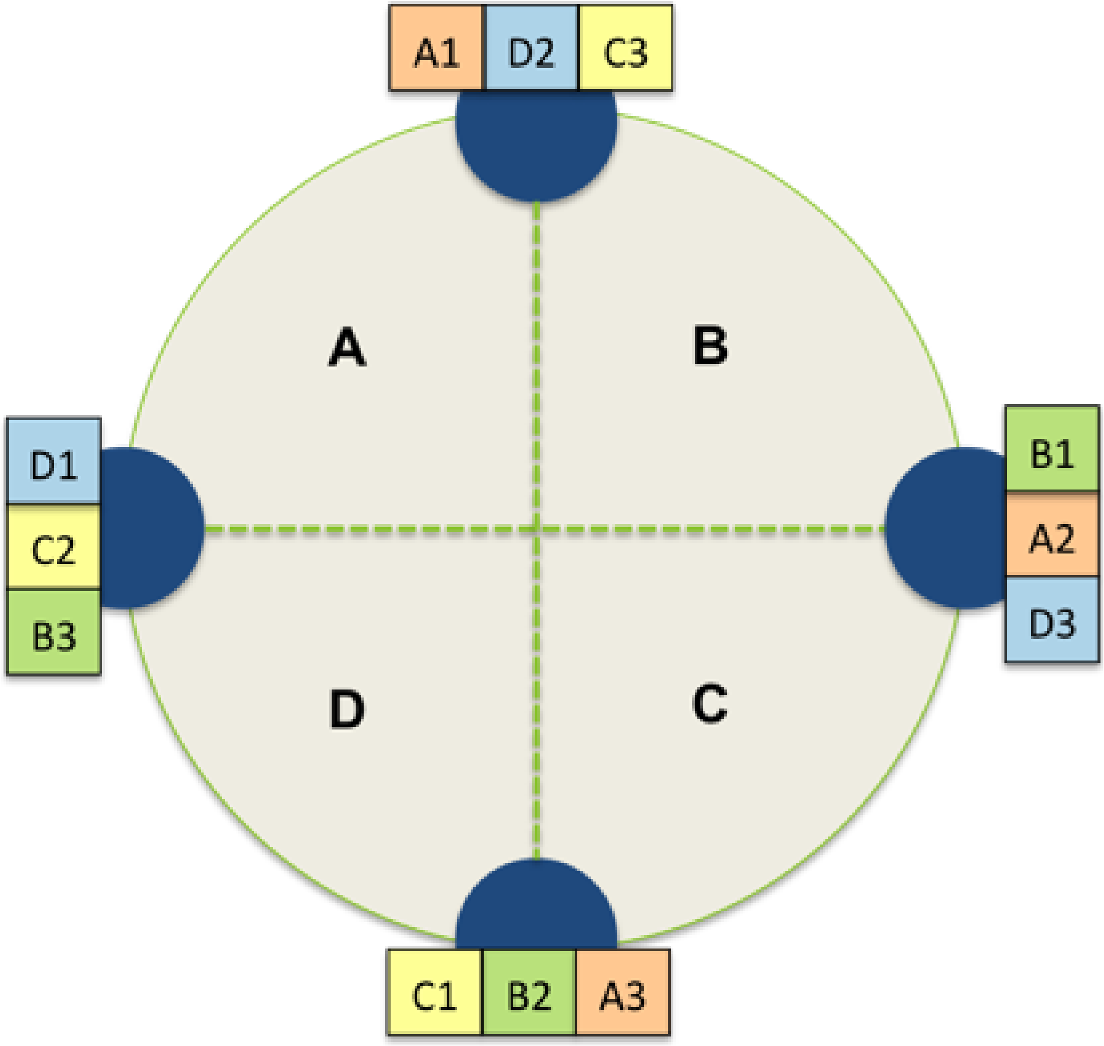
The replica placement strategy determines how replicas for a keyspace are distributed across the cluster. The replica placement strategy is set when you create a keyspace.

There are a number of strategies to choose from based on your goals and the information you have about where nodes are located.

SimpleStrategy

SimpleStrategy is the default replica placement strategy when creating a keyspace using the Cassandra CLI. Other interfaces, such as the CQL utility, require you to explicitly specify a strategy.

SimpleStrategy places the first replica on a node determined by the *partitioner*. Additional replicas are placed on the next nodes clockwise in the ring without considering rack or data center location.



NetworkTopologyStrategy

NetworkTopologyStrategy is the preferred replication placement strategy when you have information about how nodes are grouped in your data center, or you have (or plan to have) your cluster deployed across multiple data centers. This strategy allows you to specify how many replicas you want in each data center.

When deciding how many replicas to configure in each data center, the primary considerations are (1) being able to satisfy reads locally, without incurring cross-datacenter latency, and (2) failure scenarios.

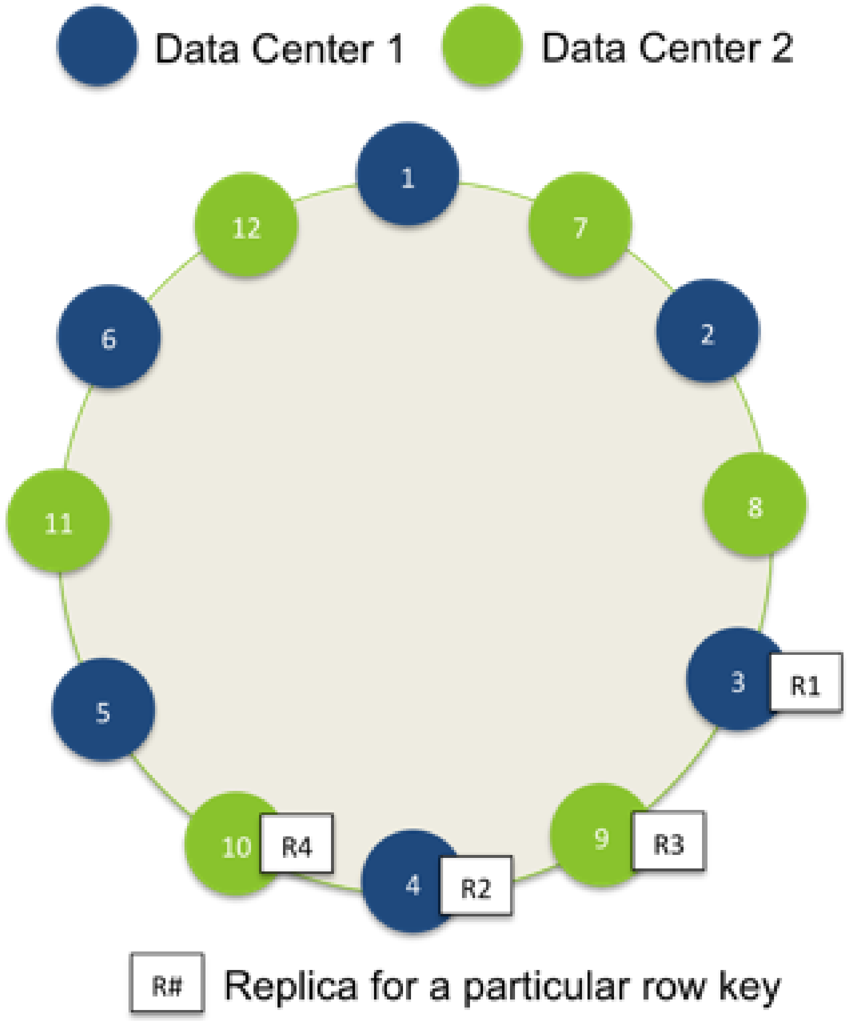
The two most common ways to configure multiple data center clusters are:

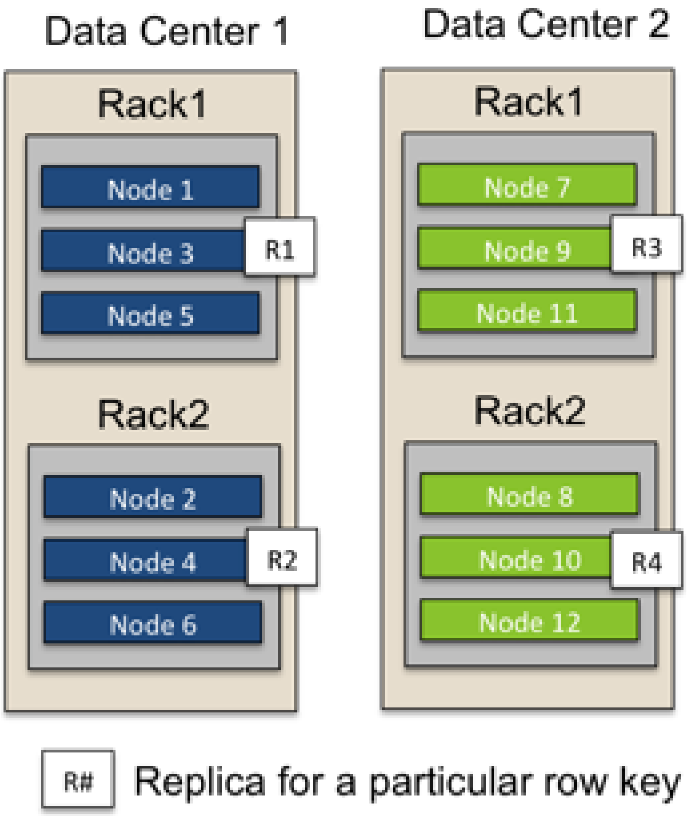
- Two replicas in each data center. This configuration tolerates the failure of a single node per replication group, and still allows local reads at a *consistency level* of ONE.
- Three replicas in each data center. This configuration tolerates the failure of a one node per replication group at a strong *consistency level* of LOCAL_QUORUM, or tolerates multiple node failures per data center using consistency level ONE.

Asymmetrical replication groupings are also possible depending on your use case. For example, you may want to have three replicas per data center to serve real-time application requests, and then have a single replica in a separate data center designated to running analytics. In Cassandra, the term *data center* is synonymous with *replication group* - it is a grouping of nodes configured together for replication purposes. It does not have to be a *physical* data center.

With NetworkTopologyStrategy, replica placement is determined independently within each data center (or replication group). The first replica per data center is placed according to the *partitioner* (same as with SimpleStrategy). Additional replicas in the same data center are then determined by walking the ring clockwise until a node in a different rack from the previous replica is found. If there is no such node, additional replicas will be placed in the same rack. NetworkTopologyStrategy prefers to place replicas on distinct racks if possible. Nodes in the same rack (or similar physical grouping) can easily fail at the same time due to power, cooling, or network issues.

Here is an example of how NetworkTopologyStrategy would place replicas spanning two data centers with a total replication factor of 4 (two replicas in Data Center 1 and two replicas in Data Center 2):


Data Center 1
Data Center 2
1
7
2
8
3
R1
9
R3
4
R2
10
R4
5
11
6
12
R#
Replica for a particular row key



Notice how tokens are assigned to alternating racks.

`NetworkTopologyStrategy` relies on a properly configured *snitch* to place replicas correctly across data centers and racks, so it is important to configure your cluster to use a snitch that can correctly determine the locations of nodes in your network.

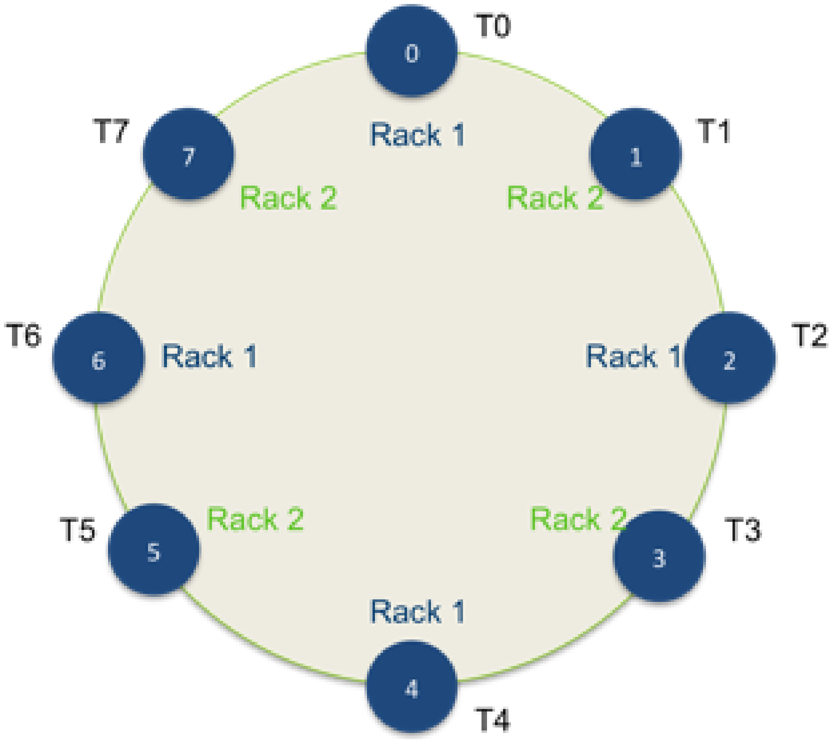
Note

`NetworkTopologyStrategy` should be used in place of the `OldNetworkTopologyStrategy`, which only supports a limited configuration of exactly 3 replicas across 2 data centers, with no control over which data center gets two replicas for any given row key. Some rows will have two replicas in the first and one in the second, while others will have two in the second and one in the first.

About Snitches

The snitch is a configurable component of a Cassandra cluster used to define how the nodes are grouped together within the overall network topology (such as rack and data center groupings). Cassandra uses this information to route inter-node requests as efficiently as possible within the confines of the replica placement strategy. The snitch does not affect requests between the client application and Cassandra (it does not control which node a client connects to).

Snitches are configured for a Cassandra cluster in the *cassandra.yaml* configuration file. All nodes in a cluster should use the same snitch configuration. When assigning tokens, assign them to alternating racks. For example: rack1, rack2, rack3, rack1, rack2, rack3, and so on.



Assign tokens to nodes in alternating racks

The following snitches are available:

SimpleSnitch

The SimpleSnitch (the default) is appropriate if you have no rack or data center information available. Single-data center deployments (or single-zone in public clouds) usually fall into this category.

If using this snitch, use `replication_factor=<#>` when defining your keyspace *strategy_options*. This snitch does not recognize data center or rack information.

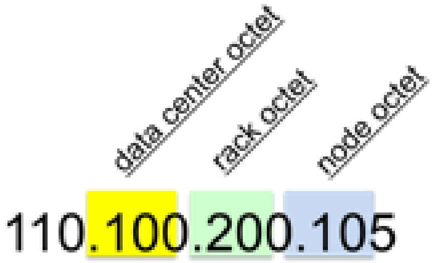
DseSimpleSnitch

DseSimpleSnitch is used in DataStax Enterprise (DSE) deployments only. It logically configures Hadoop analytics nodes in a separate data center from pure Cassandra nodes in order to segregate analytic and real-time workloads. It can be used for mixed-workload DSE clusters located in one physical data center. It can also be used for multi-data center DSE clusters that have exactly 2 data centers, with all analytic nodes in one data center and all Cassandra real-time nodes in another data center.

If using this snitch, use `Analytics` or `Cassandra` as your data center names when defining your keyspace *strategy_options*.

RackInferringSnitch

RackInferringSnitch infers the topology of the network by analyzing the node IP addresses. This snitch assumes that the second octet identifies the data center where a node is located, and the third octet identifies the rack.



If using this snitch, use the second octet number of your node IPs as your data center names when defining your keyspace *strategy_options*. For example, `100` would be the data center name.

PropertyFileSnitch

`PropertyFileSnitch` determines the location of nodes by referring to a user-defined description of the network details located in the property file `cassandra-topology.properties`. This snitch is best when your node IPs are not uniform or you have complex replication grouping requirements. See *Configuring the PropertyFileSnitch* for more information.

If using this snitch, you can define your data center names to be whatever you want. Just make sure the data center names you define in the `cassandra-topology.properties` file correlates to what you name your data centers in your keyspace *strategy_options*.

EC2Snitch

EC2Snitch is for simple cluster deployments on Amazon EC2 where all nodes in the cluster are within the same region. Instead of using the node's IP address to infer node location, this snitch uses the AWS API to request the region and availability zone of a node. The region is treated as the data center and the availability zones are treated as racks within the data center. For example, if a node is in `us-east-1a`, `us-east` is the data center name and `1a` is the rack location.

If using this snitch, use the EC2 region name (for example,``us-east``) as your data center name when defining your keyspace *strategy_options*.

EC2MultiRegionSnitch

EC2MultiRegionSnitch is for cluster deployments on Amazon EC2 where the cluster spans multiple regions. Instead of using the node's IP address to infer node location, this snitch uses the AWS API to request the region and availability zone of a node. Regions are treated as data centers and availability zones are treated as racks within a data center. For example, if a node is in `us-east-1a`, `us-east` is the data center name and `1a` is the rack location.

If using this snitch, you must configure each Cassandra node so that *listen_address* is set to the *private* IP address or the node, and *broadcast_address* is set to the *public* IP address of the node. This allows Cassandra nodes in one EC2 region to bind to nodes in another region, thus enabling multi-data center support.

If using this snitch, use the EC2 region name (for example,``us-east``) as your data center names when defining your keyspace *strategy_options*.

About Dynamic Snitching

By default, all snitches also use a dynamic snitch layer that monitors read latency and, when possible, routes requests away from poorly-performing nodes. The dynamic snitch is enabled by default, and is recommended for use in most deployments.

Dynamic snitch thresholds can be configured in the *cassandra.yaml* configuration file for a node.

About Client Requests in Cassandra

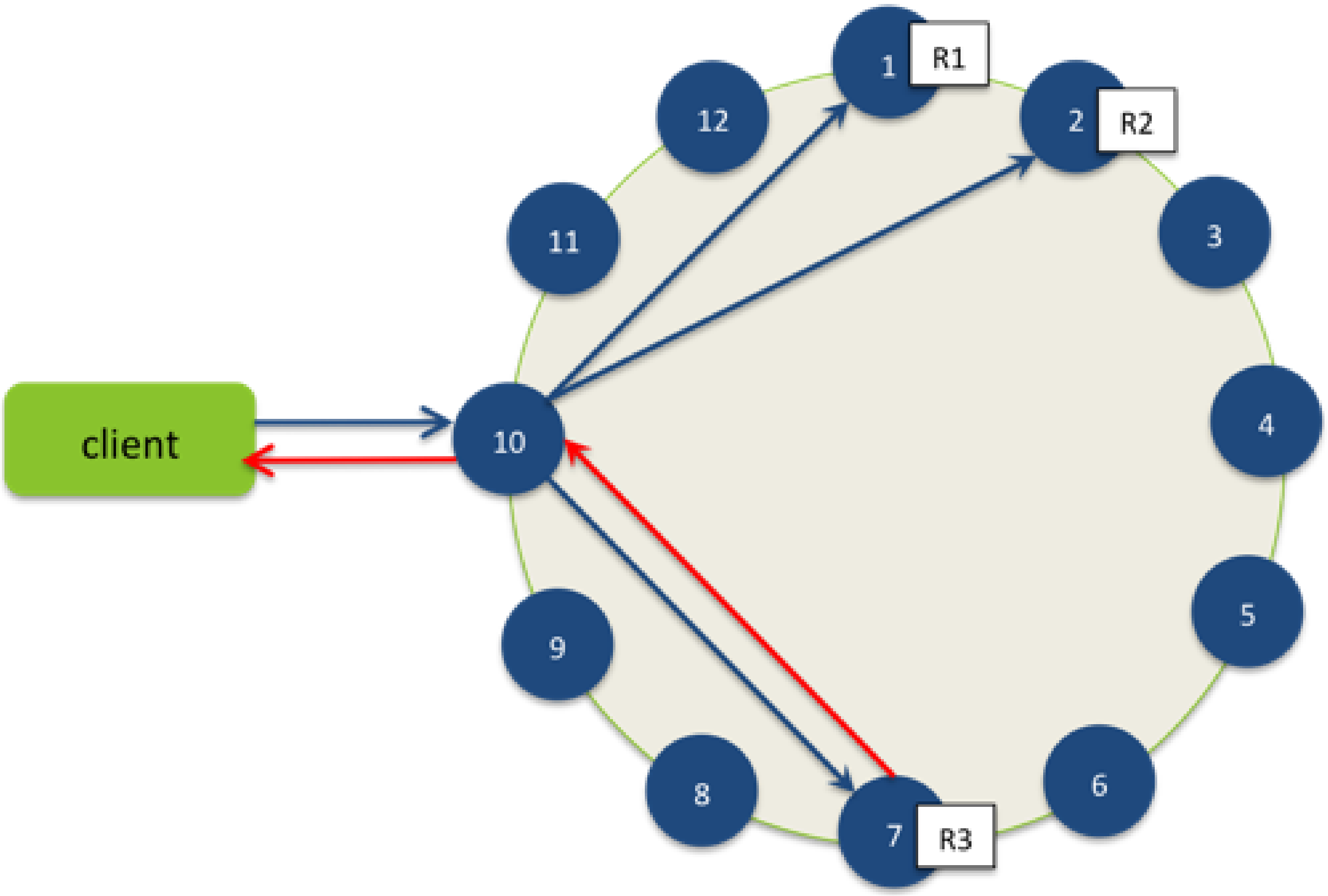
All nodes in Cassandra are peers. A client read or write request can go to any node in the cluster. When a client connects to a node and issues a read or write request, that node serves as the *coordinator* for that particular client operation.

The job of the coordinator is to act as a proxy between the client application and the nodes (or replicas) that own the data being requested. The coordinator determines which nodes in the ring should get the request based on the cluster configured *partitioner* and *replica placement strategy*.

About Write Requests

For writes, the coordinator sends the write to *all* replicas that own the row being written. As long as all replica nodes are up and available, they will get the write regardless of the *consistency level* specified by the client. The write consistency level determines how many replica nodes must respond with a success acknowledgement in order for the write to be considered successful.

For example, in a single data center 10 node cluster with a replication factor of 3, an incoming write will go to all 3 nodes that own the requested row. If the write consistency level specified by the client is ONE, the first node to complete the write responds back to the coordinator, which then proxies the success message back to the client. A consistency level of ONE means that it is possible that 2 of the 3 replicas could miss the write if they happened to be down at the time the request was made. If a replica misses a write, the row will be made consistent later via one of Cassandra's *built-in repair mechanisms*: hinted handoff, read repair or anti-entropy node repair.

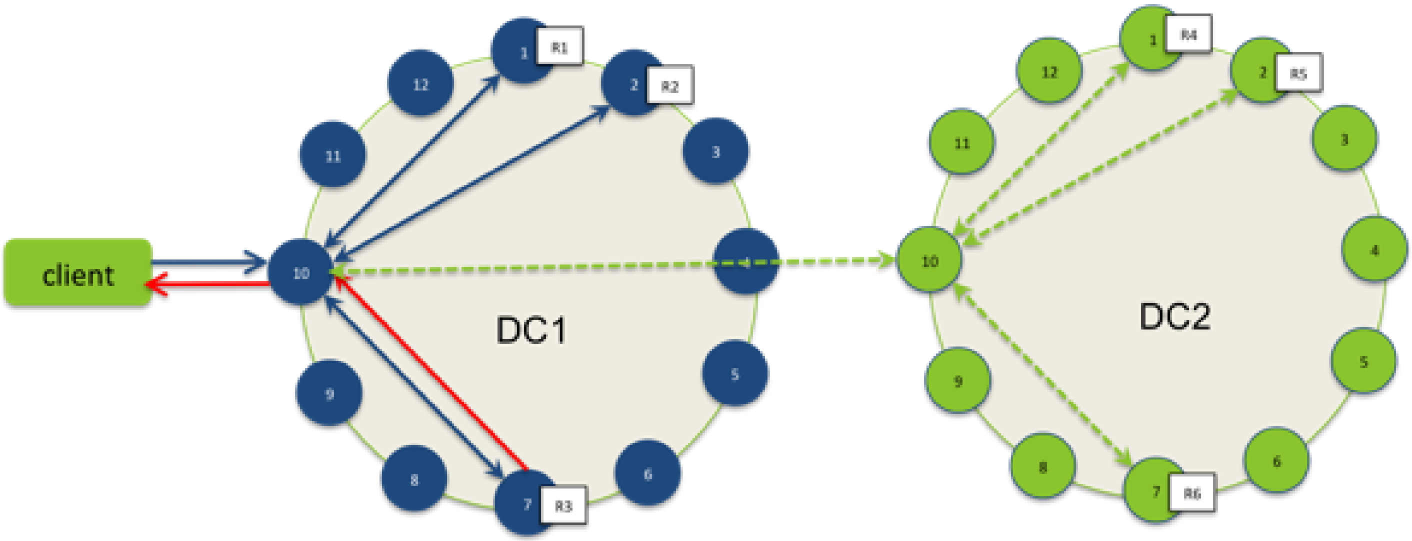


Also see *About Writes in Cassandra* for more information about how Cassandra processes writes locally at the node level.

About Multi-Data Center Write Requests

In multi data center deployments, Cassandra optimizes write performance by choosing one coordinator node in each remote data center to handle the requests to replicas within that data center. The coordinator node contacted by the client application only needs to forward the write request to one node in each remote data center.

If using a consistency level of ONE or LOCAL_QUORUM, only the nodes in the same data center as the coordinator node must respond to the client request in order for the request to succeed. This way, geographical latency does not impact client request response times.



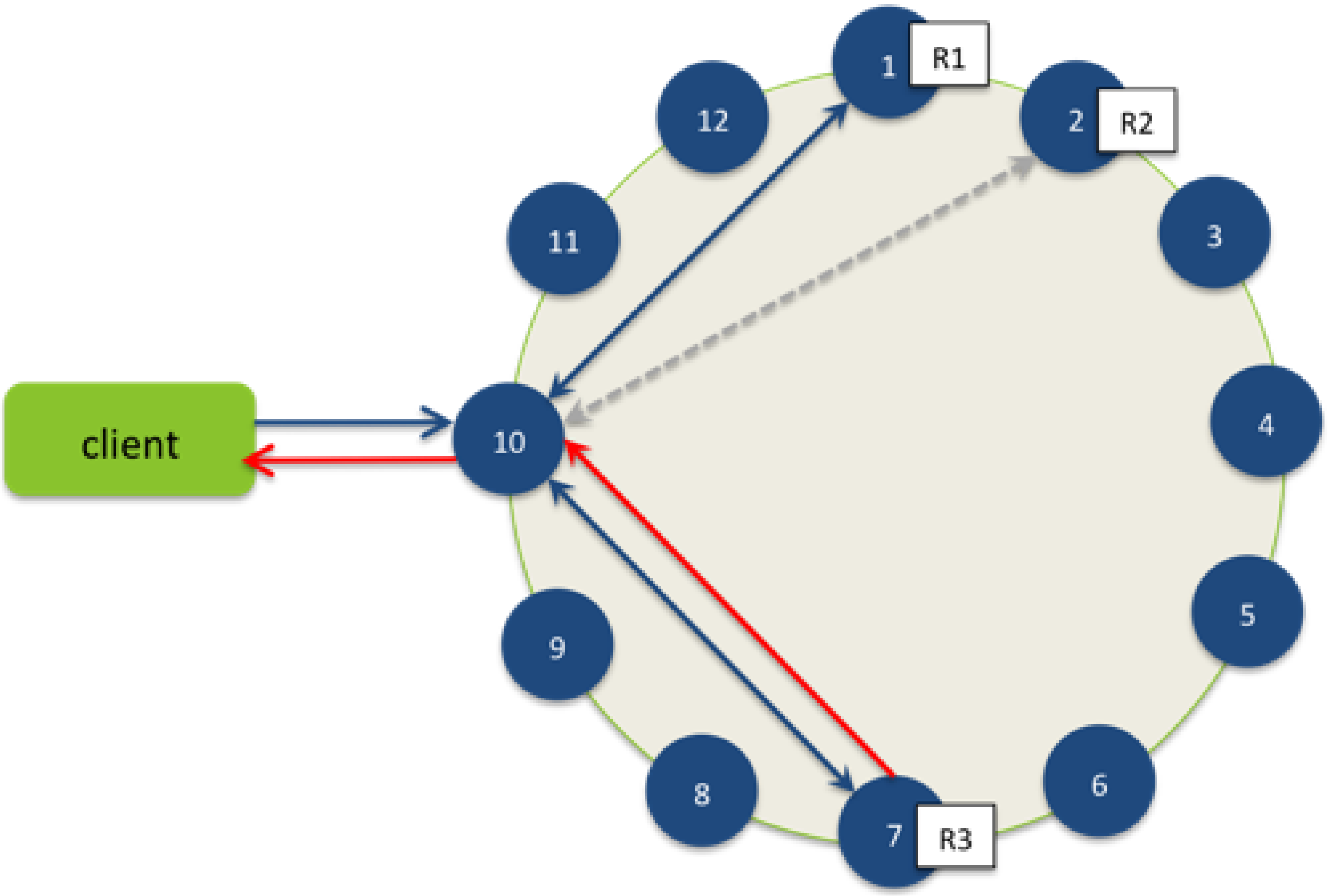
About Read Requests

For reads, there are two types of read requests that a coordinator can send to a replica; a direct read request and a background *read repair* request. The number of replicas contacted by a direct read request is determined by the consistency level specified by the client. Background read repair requests are sent to any additional replicas that did not receive a direct request. Read repair requests ensure that the requested row is made consistent on all replicas.

Thus, the coordinator first contacts the replicas specified by the consistency level. The coordinator will send these requests to the replicas that are currently responding most promptly. The nodes contacted will respond with the requested data; if multiple nodes are contacted, the rows from each replica are compared in memory to see if they are consistent. If they are not, then the replica that has the most recent data (based on the timestamp) is used by the coordinator to forward the result back to the client.

To ensure that all replicas have the most recent version of frequently-read data, the coordinator also contacts and compares the data from all the remaining replicas that own the row in the background, and if they are inconsistent, issues writes to the out-of-date replicas to update the row to reflect the most recently written values. This process is known as *read repair*. Read repair can be configured per column family (using *read_repair_chance*), and is enabled by default.

For example, in a cluster with a replication factor of 3, and a read consistency level of QUORUM, 2 of the 3 replicas for the given row are contacted to fulfill the read request. Supposing the contacted replicas had different versions of the row, the replica with the most recent version would return the requested data. In the background, the third replica is checked for consistency with the first two, and if needed, the most recent replica issues a write to the out-of-date replicas.



Also see *About Reads in Cassandra* for more information about how Cassandra processes reads locally at the node level.

Planning a Cassandra Cluster Deployment

When planning a Cassandra cluster deployment, you should first have a good idea of the initial volume of data you plan to store, as well as what your typical application workload will be.

Selecting Hardware

As with any application, choosing appropriate hardware depends on selecting the right balance of the following resources: Memory, CPU, Disk, and Network.

Memory

The more memory a Cassandra node has, the better read performance will be. More RAM allows for larger cache sizes, reducing disk I/O for reads. More RAM also allows for larger memory tables (memtables) to hold the most recently written data. Larger memtables lead to a fewer number of SSTables being flushed to disk and fewer files to scan during a read. The minimum that should be considered for a production deployment is 4GB, with 8GB-16GB being the most common, and many production clusters using 32GB or more per node. The ideal amount of RAM depends on the anticipated size of your hot data.

CPU

Insert-heavy workloads will actually be CPU-bound in Cassandra before being memory-bound. Cassandra is highly concurrent and will benefit from many CPU cores. Currently, nodes with 8 CPU cores typically provide the best price/performance ratio. On virtualized machines, consider using a cloud provider that allows CPU bursting.

Disk

When choosing disks, you should consider both capacity (how much data you plan to store) and I/O (the write/read throughput rate). Most workloads are best served by using less expensive SATA disks and scaling disk capacity and I/O by adding more nodes (with a lot of RAM).

Solid-state drives (SSDs) are also a valid alternative for Cassandra. Cassandra's sequential, streaming write patterns minimize the undesirable effects of write amplification associated with SSDs.

Cassandra persists data to disk for two different purposes: it constantly appends to a commit log for write durability, and periodically flushes in-memory data structures to SSTable data files for persistent storage of column family data. It is highly recommended to use a different disk device for the commit log than for the SSTables. The commit log does not need much capacity, but throughput should be enough to accommodate your expected write load. Data directories should be large enough to house all of your data, but with enough throughput to handle your expected (non-cached) read load and the disk I/O required by flushing and compaction.

During compaction and node repair, disk utilization can - depending on the compaction settings used - increase substantially in your data directory volume. For this reason, DataStax recommends leaving an adequate (10-50%) amount of free space available on a node.

For disk fault tolerance and data redundancy, there are two reasonable approaches:

- Use RAID0 on your data volume and rely on Cassandra replication for disk failure tolerance. If you lose a disk on a node, you can recover lost data through Cassandra's built-in repair.
- Use RAID10 to avoid large repair operations after a single disk failure.

If you have disk capacity to spare, DataStax recommends using RAID10. If disk capacity is a bottleneck for your workload, use RAID0.

Network

Since Cassandra is a distributed data store, it puts load on the network to handle read/write requests and replication of data across nodes. You want to choose reliable, redundant network interfaces and make sure that your network can handle traffic between nodes without bottlenecks.

Cassandra is efficient at routing requests to replicas that are geographically closest to the coordinator node handling the request. Cassandra will pick a replica in the same rack if possible, and will choose replicas located in the same data center over replicas in a remote data center.

Cassandra uses the following ports. If using a firewall, make sure that nodes within a cluster can reach each other on these ports:

Port	Description
7000	Cassandra intra-node communication port
9160	Thrift client port
7199	JMX monitoring port (8080 in prior releases)

Planning an Amazon EC2 Cluster

Cassandra clusters can be deployed on cloud infrastructures such as Amazon EC2.

For production Cassandra clusters on EC2, use L or XL instances with local storage. RAID0 the ephemeral disks, and put both the data directory and the commit log on that volume. This has proved to be better in practice than putting the commit log on the root volume (which is also a shared resource). For data redundancy, consider deploying your Cassandra cluster across multiple availability zones or using EBS volumes to store your Cassandra backup files.

EBS volumes are *not* recommended for Cassandra data volumes - their network performance and disk I/O are not good fits for Cassandra for the following reasons:

- EBS volumes contend directly for network throughput with standard packets. This means that EBS throughput is likely to fail if you saturate a network link.
- EBS volumes have unreliable performance. I/O performance can be exceptionally slow, causing the system to backload reads and writes until the entire cluster becomes unresponsive.
- Adding capacity by increasing the number of EBS volumes per host does not scale. You can easily surpass the ability of the system to keep effective buffer caches and concurrently serve requests for all of the data it is responsible for managing.

DataStax provides an Amazon Machine Image (AMI) to allow you to quickly deploy a multi-node Cassandra cluster on Amazon EC2. The DataStax AMI initializes all nodes in one availability zone using the *SimpleSnitch*.

If you want an EC2 cluster that spans multiple regions and availability zones, do not use the DataStax AMI. Instead, initialize your EC2 instances for each Cassandra node and then configure the cluster as a multi data center cluster.

Capacity Planning

The estimates in this section can be used as guidelines for planning the size of your Cassandra cluster based on the data you plan to store. To estimate capacity, you should first have a good understanding of the sizing of the raw data you plan to store, and how you plan to model your data in Cassandra (number of column families, rows, columns per row, and so on).

Calculating Usable Disk Capacity

To calculate how much data your Cassandra nodes can hold, calculate the usable disk capacity per node and then multiply that by the number of nodes in your cluster. Remember that in a production cluster, you will typically have your commit log and data directories on different disks. This calculation is for estimating the usable capacity of the data volume.

Start with the raw capacity of the physical disks:

```
raw_capacity = 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
```

During normal operations, Cassandra routinely requires disk capacity for compaction and repair operations. For optimal performance and cluster health, DataStax recommends that you do not fill your disks to capacity, but run at 50-80 percent capacity. With this in mind, calculate the usable disk space as follows (example below uses 50%):

```
formatted_disk_space * 0.5 = usable_disk_space
```

Calculating User Data Size

As with all data storage systems, the size of your raw data will be larger once it is loaded into Cassandra due to storage overhead. On average, raw data will be about 2 times larger on disk after it is loaded into the database, but could be much smaller or larger depending on the characteristics of your data and column families. The calculations in this section account for data persisted to disk, not for data stored in memory.

- **Column Overhead** - Every column in Cassandra incurs 15 bytes of overhead. Since each row in a column family can have different column names as well as differing numbers of columns, metadata is stored for *each* column. For counter columns and expiring columns, add an additional 8 bytes (23 bytes column overhead). So the total size of a *regular* column is:

  ```
  total_column_size = column_name_size + column_value_size + 15
  ```

- **Row Overhead** - Just like columns, every row also incurs some overhead when stored on disk. Every row in Cassandra incurs 23 bytes of overhead.

- **Primary Key Index** - Every column family also maintains a primary index of its row keys. Primary index overhead becomes more significant when you have lots of *skinny* rows. Sizing of the primary row key index can be estimated as follows (in bytes):

  ```
  primary_key_index = number_of_rows * (32 + average_key_size)
  ```

- **Replication Overhead** - The replication factor obviously plays a role in how much disk capacity is used. For a replication factor of 1, there is no overhead for replicas (as only one copy of your data is stored in the cluster). If replication factor is greater than 1, then your total data storage requirement will include replication overhead.

  ```
  replication_overhead = total_data_size * (replication_factor - 1)
  ```

Choosing Node Configuration Options

A major part of planning your Cassandra cluster deployment is understanding and setting the various node configuration properties. This section explains the various configuration decisions that need to be made before deploying a Cassandra cluster, be it a single-node, multi-node, or multi-data center cluster.

These properties mentioned in this section are set in the *cassandra.yaml* configuration file. Each node should be correctly configured before starting it for the first time.

Storage Settings

By default, a node is configured to store the data it manages in /var/lib/cassandra. In a production cluster deployment, you should change the *commitlog_directory* so it is on a different disk device than the *data_file_directories*.

Gossip Settings

The gossip settings control a nodes participation in a cluster and how the node is known to the cluster.

Property	Description
cluster_name	Name of the cluster that this node is joining. Should be the same for every node in the cluster.
listen_address	The IP address or hostname that other Cassandra nodes will use to connect to this node. Should be changed from localhost to the public address for the host.
seeds	A comma-delimited list of node IP addresses used to bootstrap the gossip process. Every node should have the same list of seeds. In multi data center clusters, the seed list should include a node from *each* data center.
storage_port	The intra-node communication port (default is 7000). Should be the same for every node in the cluster.
initial_token	The initial token is used to determine the range of data this node is responsible for.

Purging Gossip State on a Node

Gossip information is also persisted locally by each node to use immediately next restart without having to wait for gossip. To clear gossip history on node restart (for example, if node IP addresses have changed), add the following line to the cassandra-env.sh file. This file is located in /usr/share/cassandra or $CASSANDRA_HOME/conf in Cassandra installations. This file is located in /etc/brisk/cassandra or $BRISK_HOME/resources/cassandra/conf in Brisk installations.

```
-Dcassandra.load_ring_state=false
```

Partitioner Settings

When you deploy a Cassandra cluster, you need to make sure that each node is responsible for roughly an equal amount of data. This is also known as load balancing. This is done by configuring the *partitioner* for each node, and correctly assigning the node an *initial_token* value.

DataStax strongly recommends using the *RandomPartitioner* (the default) for all cluster deployments. Assuming use of this partitioner, each node in the cluster is assigned a *token* that represents a hash value within the range of 0 to 2**127.

For clusters where all nodes are in a single data center, you can calculate tokens by dividing the range by the total number of nodes in the cluster. In multi-data center deployments, tokens should be calculated such that each data center is individually load balanced as well. See *Calculating Tokens* for the different approaches to generating tokens for nodes in single and multi-data center clusters.

Snitch Settings

The snitch is responsible for knowing the location of nodes within your network topology. This affects where replicas are placed as well as how requests are routed between replicas. The *endpoint_snitch* property configures the snitch for a node. All nodes should have the exact same snitch configuration.

For a single data center (or single node) cluster, using the default *SimpleSnitch* is usually sufficient. However, if you plan to expand your cluster at a later time to multiple racks and data centers, it will be easier if you choose a rack and data center aware snitch from the start. All *snitches* are compatible with all *replica placement strategies*.

Configuring the PropertyFileSnitch

The `PropertyFileSnitch` requires you to define network details for each node in the cluster in a `cassandra-topology.properties` configuration file. A sample of this file is located in `/etc/cassandra/conf/cassandra.yaml` in packaged installations or `$CASSANDRA_HOME/conf/cassandra.yaml` in binary installations.

Every node in the cluster should be described in this file, and this file should be exactly the same on every node in the cluster if you are using the `PropertyFileSnitch`.

For example, supposing you had non-uniform IPs and two physical data centers with two racks in each, and a third logical data center for replicating analytics data:

```
# Data Center One

175.56.12.105=DC1:RAC1
175.50.13.200=DC1:RAC1
175.54.35.197=DC1:RAC1

120.53.24.101=DC1:RAC2
120.55.16.200=DC1:RAC2
120.57.102.103=DC1:RAC2

# Data Center Two

110.56.12.120=DC2:RAC1
110.50.13.201=DC2:RAC1
110.54.35.184=DC2:RAC1

50.33.23.120=DC2:RAC2
50.45.14.220=DC2:RAC2
50.17.10.203=DC2:RAC2

# Analytics Replication Group

172.106.12.120=DC3:RAC1
172.106.12.121=DC3:RAC1
172.106.12.122=DC3:RAC1
```

```
# default for unknown nodes
default=DC3:RAC1
```

If using this snitch, you can define your data center and rack names to be whatever you want. Just make sure the data center names you define in the `cassandra-topology.properties` file correlates to what you name your data centers in your keyspace *strategy_options*.

Choosing Keyspace Replication Options

When you create a keyspace, you must define the *replica placement strategy* and the number of replicas you want. DataStax recommends always choosing `NetworkTopologyStrategy` for both single and multi-data center clusters. It is as easy to use as `SimpleStrategy` and allows for expansion to multiple data centers in the future, should that become useful. It is much easier to configure the most flexible replication strategy up front, than to reconfigure replication after you have already loaded data into your cluster.

`NetworkTopologyStrategy` takes as options the number of replicas you want *per data center*. Even for single data center (or single node) clusters, you can use this replica placement strategy and just define the number of replicas for one data center. For example (using `cassandra-cli`):

```
[default@unknown] CREATE KEYSPACE test
WITH placement_strategy = 'NetworkTopologyStrategy'
AND strategy_options=[{us-east:6}];
```

Or for a multi-data center cluster:

```
[default@unknown] CREATE KEYSPACE test
WITH placement_strategy = 'NetworkTopologyStrategy'
AND strategy_options=[{DC1:6,DC2:6,DC3:3}];
```

When declaring the keyspace *strategy_options*, what you name your data centers depends on the *snitch* you have chosen for your cluster. The data center names must correlate to the snitch you are using in order for replicas to be placed in the correct location.

As a general rule, the number of replicas should not exceed the number of nodes in a replication group. However, it is possible to increase the number of replicas, and then add the desired number of nodes afterwards. When the replication factor exceeds the number of nodes, writes will be rejected, but reads will still be served as long as the desired *consistency level* can be met.

Installing and Initializing a Cassandra Cluster

Installing and initializing a Cassandra cluster involves installing the Cassandra software on each node, and configuring each node so that it is prepared to join the cluster (see *Planning a Cassandra Cluster Deployment* for considerations on choosing the right configuration options for your environment). After each node is installed and configured, start each node sequentially beginning with the seed node(s).

Installing Cassandra Using the Packaged Releases

DataStax Community Edition provides RedHat/CentOS and Debian/Ubuntu packaged releases for Cassandra on Linux. `rpm` and `deb` packages are currently supported through the `yum` and `apt` package management tools.

DataStax also provides a GUI Windows installation package for Microsoft Windows and a tar package for Mac. For more information on Windows and Mac installations, see the *DataStax Community Install Instructions*.

What follows are instructions on installing Cassandra on Linux using DataStax supplied packages.

Creating the Cassandra User and Configuring sudo

The packages install and run Cassandra as the `cassandra` user. The best practice is to create the `cassandra` user and add that user to the `sodoers` list. Then install and run as the `cassandra` user using `sudo`. For example (as root):

1. Add the `cassandra` user and set its password:

   ```
   # useradd cassandra

   # passwd cassandra
   ```

2. Edit `/etc/sudoers` and add the following line:

   ```
   cassandra ALL=(ALL)    ALL
   ```

3. Change to the Cassandra user:

   ```
   # su - cassandra
   ```

4. Proceed with your package installation.

Installing Cassandra RPM Packages

DataStax provides `yum` repositories for CentOS and RedHat Enterprise Linux 5 and 6 and Fedora 12, 13 and 14. These instructions assume that you have the `yum` package management application installed, and that you have sudo (or root) access on the machine where you are installing.

Note

By downloading community software from DataStax you agree to the terms of the DataStax Community EULA (end user license agreement) posted on the DataStax web site.

1. Make sure you have EPEL (Extra Packages for Enterprise Linux) installed. EPEL contains dependent packages required by DSE, such as `jna` and `jpackage-utils`:

```
$ sudo rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/i386/epel-release-5-4.noarch.rpm
```

2. Add a `yum` repository specification for the DataStax repository in `/etc/yum.repos.d`. For example:

   ```
   $ sudo vi /etc/yum.repos.d/datastax.repo
   ```

3. In this file add the following lines for the DataStax repository:

   ```
   [datastax]
   name= DataStax Repo for Apache Cassandra
   baseurl=http://rpm.datastax.com/community
   enabled=1
   gpgcheck=0
   ```

4. Install the package using `yum`.

   ```
   $ sudo yum install dsc
   ```

This installs the Cassandra, DataStax Community demos, and OpsCenter packages.

Installing Sun JRE on RedHat Systems

DataStax recommends installing the most recently released version of the Oracle Sun Java Runtime Environment (JRE), also referred to as the Java Virtual Machine (JVM). Versions earlier than 1.6.0_19 should not be used.

The rpm packages install the OpenJDK Java Runtime Environment (JRE) instead of the Oracle Sun JRE. After installing using the rpm packaged releases, configure your operating system to use the Oracle Sun JRE instead of OpenJDK.

1. Check which version of the JRE your system is using. If your system is using the OpenJDK Runtime Environment, you will need to change it to use the Oracle Sun JRE.

   ```
   $ java -version
   ```

2. Go to the Oracle Java Runtime Environment Download Page, accept the license agreement, and download the `Linux x64-RPM` Installer or `Linuxx86-RPM` Installer (depending on your platform).
3. Go to the directory where you downloaded the JRE package, and change the permissions so the file is executable. For example:

```
$ cd /tmp
$ chmod a+x jre-6u25-linux-x64-rpm.bin
```

4. Extract and run the rpm file. For example:

```
$ sudo ./jre-6u25-linux-x64-rpm.bin
```

The rpm installs the JRE into `/usr/java/`.

5. Configure your system so that it is using the Oracle Sun JRE instead of the OpenJDK JRE. Use the `alternatives` command to add a symbolic link to the Oracle Sun JRE installation. For example:

```
$ sudo alternatives --install /usr/bin/java java /usr/java/jre1.6.0_25/bin/java 20000
```

6. Make sure your system is now using the correct JRE. For example:

```
$ java -version
  java version "1.6.0_25"
  Java(TM) SE Runtime Environment (build 1.6.0_25-b06)
  Java HotSpot(TM) 64-Bit Server VM (build 20.0-b11, mixed mode)
```

7. If the OpenJDK JRE is still being used, use the `alternatives` command to switch it. For example:

```
$ sudo alternatives --config java
There are 2 programs which provide 'java'.

Selection      Command
-----------------------------------------------
   1           /usr/lib/jvm/jre-1.6.0-openjdk.x86_64/bin/java
*+ 2           /usr/java/jre1.6.0_25/bin/java

Enter to keep the current selection[+], or type selection number: 2
```

Installing Cassandra Debian Packages

DataStax provides a debian package repository for Apache Cassandra. These instructions assume that you have the `aptitude` package management application installed, and that you have root access on the machine where you are installing.

Note

By downloading community software from DataStax you agree to the terms of the DataStax Community EULA (end user license agreement) posted on the DataStax web site.

1. Edit the aptitude repository source list file (`/etc/apt/sources.list`).

```
$ sudo vi /etc/apt/sources.list
```

2. In this file, add the DataStax Community repository.

   ```
   deb http://debian.datastax.com/community stable main
   ```

   (Debian Systems Only) Find the line that describes your source repository for Debian and add `contrib non-free` to the end of the line. This will allow installation of the Oracle Sun JVM instead of the OpenJDK JVM. For example:

   ```
   deb http://some.debian.mirror/debian/ $distro main contrib non-free
   ```

   Save and close the file when you are done adding/editing your sources.

3. Add the DataStax repository key to your aptitude trusted keys.

   ```
   $ wget -O - http://debian.datastax.com/debian/repo_key | sudo apt-key add -
   ```

4. Install the package.

   ```
   $ sudo apt-get update
   $ sudo apt-get install dsc
   ```

   This installs the Cassandra, DataStax Community demos, and OpsCenter packages.

6. Check which version of the Java Runtime Environment (JRE) your system is using. If your system is using the OpenJDK Runtime Environment, you will need to change it to use the Oracle Sun JRE.

   ```
   $ java -version
   ```

7. By default, the Debian packages start the Cassandra service automatically. To stop the service and clear the initial gossip history that gets populated by this initial start:

   ```
   $ sudo service cassandra stop
   $ sudo bash -c 'rm /var/lib/cassandra/data/system/*'
   ```

Installing Sun JRE on Ubuntu Systems

DataStax recommends installing the most recently released version of the Oracle Sun Java Runtime Environment (JRE), also referred to as the Java Virtual Machine (JVM). Versions earlier than 1.6.0_19 should not be used.

1. Edit the aptitude repository source list file (`/etc/apt/sources.list`).

   ```
   $ sudo vi /etc/apt/sources.list
   ```

2. Add the following repository for your operating system. For example, where <OSType> is `lenny`, `lucid`, `maverick` or `squeeze`:

   ```
   deb http://archive.canonical.com/ <OSType> partner
   ```

3. Install the Sun JRE packages:

   ```
   $ sudo apt-get update
   $ sudo apt-get install sun-java6-jre
   ```

4. A license screen will appear and prompt you to accept the license terms. Once you click **OK** and **Yes** to accept the license terms, the JRE will finish installing.

5. Configure your system so that it is using the Oracle Sun JRE instead of the OpenJDK JRE.

   ```
   $ sudo update-alternatives --config java
   ```

6. Make sure your system is now using the correct JRE. For example:

   ```
   $ sudo java -version
   java version "1.6.0_26"
   Java(TM) SE Runtime Environment (build 1.6.0_26-b03)
   Java HotSpot(TM) 64-Bit Server VM (build 20.1-b02, mixed mode)
   ```

About Packaged Installs

The packaged releases install into the following directories. The packaged releases create a `cassandra` user. When starting Cassandra as a service, the service runs as this user.

- `/var/lib/cassandra` (data directories)
- `/var/log/cassandra` (log directory)
- `/var/run/cassandra` (runtime files)
- `/usr/share/cassandra` (environment settings)
- `/usr/share/dse-demos` (DataStax demo application)
- `/usr/share/cassandra/lib` (jar files)
- `/usr/bin` (binary files)
- `/usr/sbin`
- `/etc/cassandra` (configuration files)
- `/etc/init.d` (service startup script)
- `/etc/security/limits.d` (cassandra user limits)
- `/etc/default`

Next Steps

For next steps see *Configuring and Starting a Cassandra Cluster*.

Installing the Cassandra Tarball Distribution

Binary tarball distributions of Cassandra are available from the DataStax Downloads Site. DataStax also provides a GUI Windows installation package for Microsoft Windows and a tar package for Mac. For more information on Windows and Mac installations, see the *DataStax Community Install Instructions*.

Instructions for installing Cassandra on Linux

To run Cassandra, you will need to install a Java Virtual Machine (JVM). DataStax recommends installing the most recently released version of the Sun JVM. Versions earlier than 1.6.0_19 are specifically *not* recommended. See *Installing Sun JRE on Ubuntu Systems* and *Installing Sun JRE on RedHat Systems* for instructions.

Note

By downloading community software from DataStax you agree to the terms of the DataStax Community EULA (End User License Agreement) posted on the DataStax web site.

1. Download the distribution to a location on your machine and unpack it. For example, to download and unpack Cassandra DataStax Community, you would enter the following command:

   ```
   $ wget http://downloads.datastax.com/community/dsc.tar.gz

   $ tar -xvzf dsc.tar.gz
   ```

2. For convenience, set the following environment variables in your user environment (such as .bashrc):

```
export CASSANDRA_HOME=<install_location>/<dse_package_name>

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

3. Create the data and logging directories needed by Cassandra. By default, Cassandra uses `/var/lib/cassandra` and `/var/log/cassandra`. To create these directories, run the following commands where `$USER` is the user that will run Cassandra:

```
# mkdir /var/lib/cassandra
# mkdir /var/log/cassandra
# chown -R $USER:$GROUP /var/lib/cassandra
# chown -R $USER:$GROUP /var/log/cassandra
```

Note

For information about installing the Portfolio Demo Sample Application, see *Installing a Single-Node Instance of Cassandra*.

About Cassandra Binary Installations

The following directories are installed in $CASSANDRA_HOME

- `bin` (utilities and start scripts)
- `conf` (configuration files and environment settings)
- `interface` (Thrift and Avro client APIs)
- `javadoc` (Cassandra Java API documentation)
- `lib` (jar files and license files)

Installing JNA

Installing JNA (Java Native Access) on Linux platforms can improve Cassandra memory usage. With JNA installed and configured as described in this section, Linux does not swap out the JVM, and thus avoids related performance issues.

1. Download `jna.jar` from the JNA project site.
2. Add `jna.jar` to `$CASSANDRA_HOME/lib/` or otherwise place it on the `CLASSPATH`.
3. Edit the file `/etc/security/limits.conf`, adding the following entries for the user or group that runs Cassandra:

```
$USER soft memlock unlimited
$USER hard memlock unlimited
```

Next Steps

For next steps see *Configuring and Starting a Cassandra Cluster*.

Initializing a Cassandra Cluster on Amazon EC2 Using the DataStax AMI

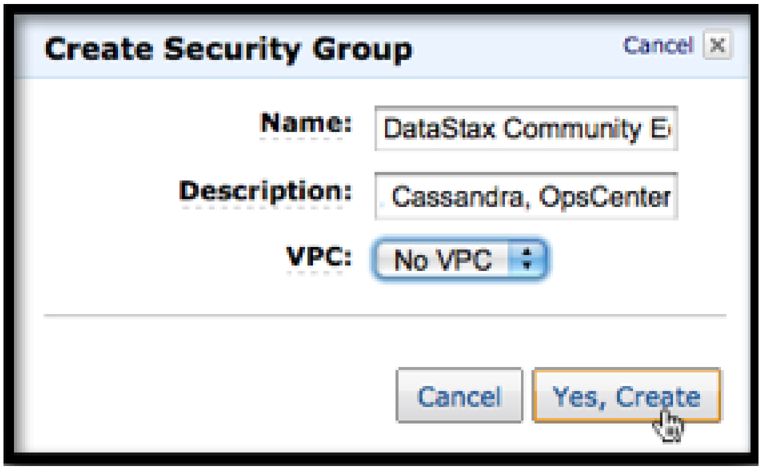
This is a step-by-step guide to using the Amazon Web Services EC2 Management Console to set up a simple Cassandra cluster using the DataStax Community Edition AMI (Amazon Machine Image). Installing via the AMI allows you to quickly deploy a Cassandra cluster within a single availability zone. When you launch the AMI, you can specify the total number of nodes in your cluster.

The DataStax Cassandra AMI does the following:

- Installs Cassandra on an Ubuntu 10.10 image
- Uses RAID0 ephemeral disks for data storage and commit log
- Uses the private interface for intra-cluster communication
- Configures a Cassandra cluster using the RandomPartitioner
- Configures the Cassandra replication strategy using the `EC2Snitch`
- Configures the seed node cluster-wide
- Starts Cassandra on all the nodes
- Installs DataStax OpsCenter on the first node in the cluster (by default)

Creating an EC2 Security Group for DataStax Community Edition

1. In your Amazon EC2 Console Dashboard, select **Security Groups** in the **Network & Security** section.
2. Click **Create Security Group**. Fill out the name and description and click **Yes, Create**.



Create Security Group

Name: DataStax Community E

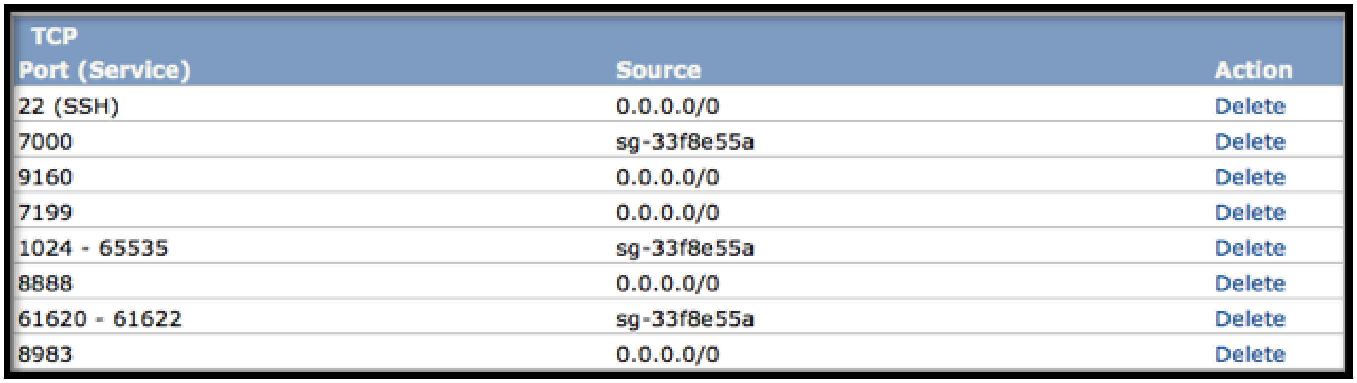
Description: Cassandra, OpsCenter

VPC: No VPC

3. Click **Inbound** and add rules for the following ports.

Port	Rule Type	Description
22	SSH	Default SSH port
7000	Custom TCP Rule	Cassandra intra-node port (source is the current security group)
9160	Custom TCP Rule	Cassandra client port
7199	Custom TCP Rule	Cassandra JMX monitoring port
1024+	Custom TCP Rule	JMX reconnection/loopback ports (source is the current security group)
8888	Custom TCP Rule	OpsCenter website port
61620	Custom TCP Rule	OpsCenter intra-node monitoring port (source is the current security group)
61621	Custom TCP Rule	OpsCenter agent port (source is the current security group)
8983	Custom TCP Rule	DataStax demo application website port

4. After you are done adding the above port rules, click **Apply Rule Changes**. Your completed port rules should look something like this:



TCP Port (Service)	Source	Action
22 (SSH)	0.0.0.0/0	Delete
7000	sg-33f8e55a	Delete
9160	0.0.0.0/0	Delete
7199	0.0.0.0/0	Delete
1024 - 65535	sg-33f8e55a	Delete
8888	0.0.0.0/0	Delete
61620 - 61622	sg-33f8e55a	Delete
8983	0.0.0.0/0	Delete

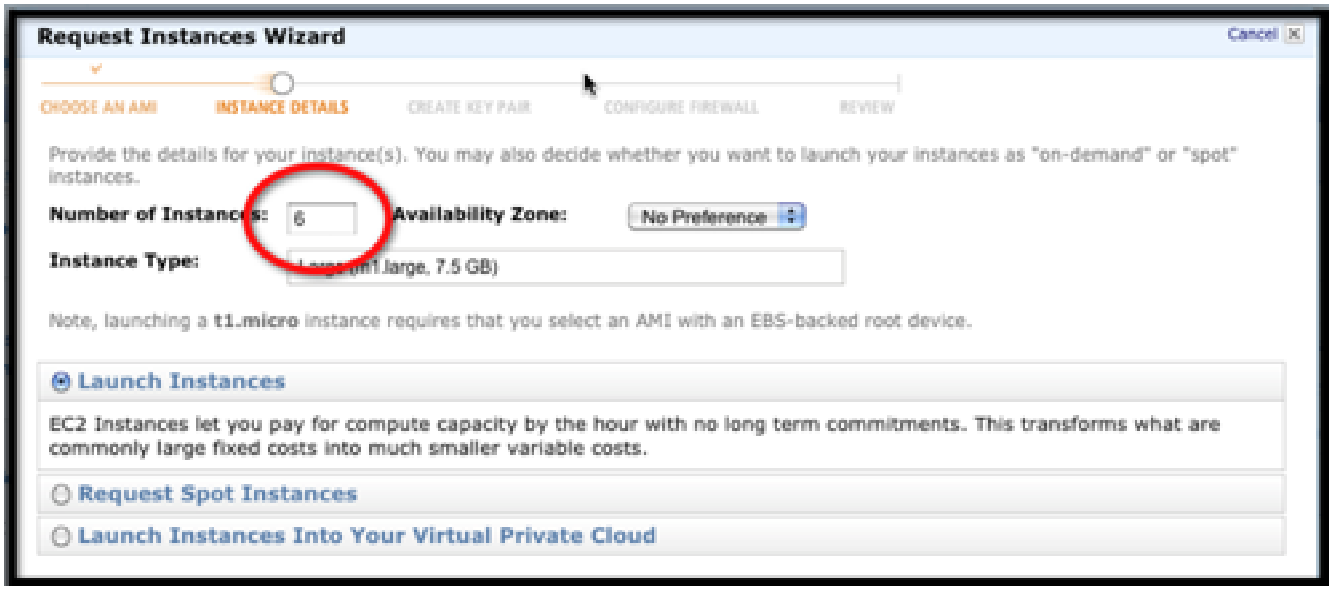
Note

This security configuration shown in this example opens up all externally accessible ports to incoming traffic from any IP address (0.0.0.0/0). If you desire a more secure configuration, see the Amazon EC2 help on Security Groups for more information on how to configure more limited access to your cluster.

Launching the DataStax Community AMI

After you have created your security group, you are ready to launch an instance of Cassandra using the DataStax Community Edition AMI.

1. From your Amazon EC2 Console Dashboard, click **Launch Instance**. Select **Launch Classic Wizard** and **Continue**.
2. On the **Choose an AMI** page, select the **Community AMIs** tab. Find **DataStax AMI, version 2.1** and click **Select** to launch it.
3. On the **Instance Details** page, enter the total number of nodes you want in your cluster in the **Number of Instances** field and select the **Instance Type**.



Provide the details for your instance(s). You may also decide whether you want to launch your instances as "on-demand" or "spot" instances.

Number of Instances: 6 Availability Zone: No Preference

Instance Type: Large (m1.large, 7.5 GB)

Note, launching a **t1.micro** instance requires that you select an AMI with an EBS-backed root device.

● Launch Instances

EC2 Instances let you pay for compute capacity by the hour with no long term commitments. This transforms what are commonly large fixed costs into much smaller variable costs.

○ Request Spot Instances

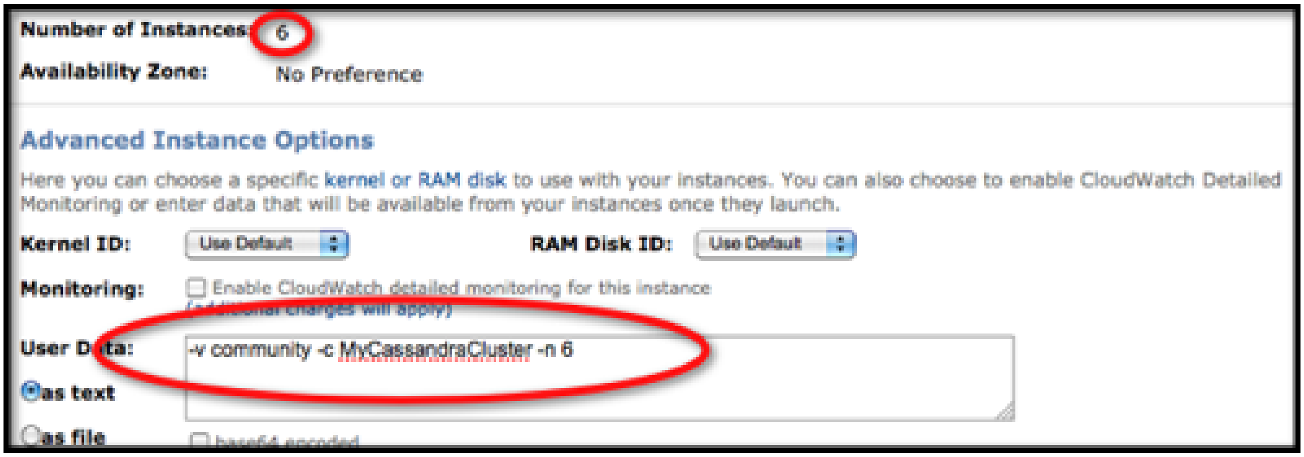
○ Launch Instances Into Your Virtual Private Cloud

4. Click **Continue**.

5. Under **Advanced Instance Options** add the following options to the **User Data** section depending on the type of cluster you want. Option parameters cannot have spaces.

   For new Cassandra clusters the available options are:

Option	Description
`-c | --clustername <name>`	Required. The name of the cluster.
`-n | --totalnodes <num_nodes>`	Required. The total number of nodes in the cluster.
`-v | --version [enterprise | community]`	Required. The version of the cluster. Use `community` to install the latest version of DataStax Community Edition.
`-o | --opscenter [no]`	Optional. By default, DataStax OpsCenter will be installed on the first instance unless you specify this option with `no`.
`-e | --email <smtp>:<port>:<email>:<password>`	Optional. Sends logs from AMI install to this email address. For example: smtp.gmail.com:587:ec2@datastax.com:pa$$word



Number of Instances: 6

Availability Zone: No Preference

Advanced Instance Options

Here you can choose a specific kernel or RAM disk to use with your instances. You can also choose to enable CloudWatch Detailed Monitoring or enter data that will be available from your instances once they launch.

Kernel ID: Use Default

RAM Disk ID: Use Default

Monitoring: □ Enable CloudWatch detailed monitoring for this instance (additional charges will apply)

User Data: -v community -c MyCassandraCluster -n 6

● as text

○ as file □ base64 encoded

6. Click **Continue**.
7. On the **Tags** page, give a name to your instance. This can be any name you like (For example: `cassandra-node`). Click **Continue**.
8. On the **Create Key Pair** page create a new key pair or select an existing key pair and click **Continue**. You will need this key (.pem file) to log in to your Cassandra nodes, so save it to a location on your local machine.
9. On the **Configure Firewall** page, select the security group you created earlier and click **Continue**.
10. On the **Review** page, review your cluster configuration and then click **Launch**.
11. Go to the **My Instances** page to see the status of your instance. Once a node has a status of **running**, you are able to connect to it.

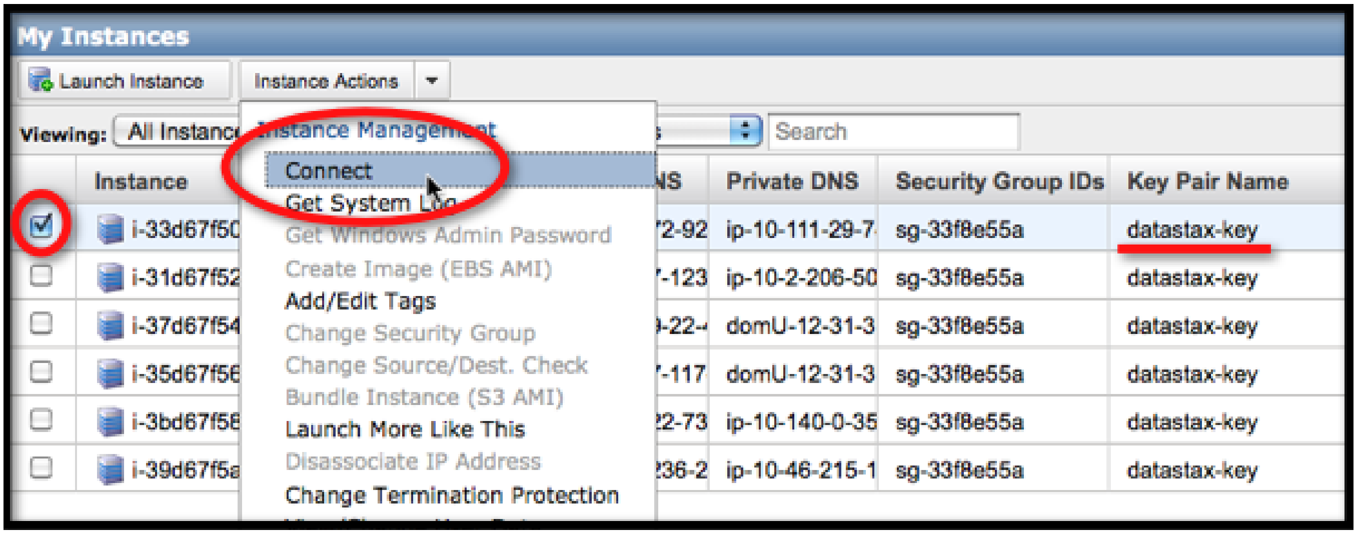
Connecting to Your Cassandra EC2 Instance

You can connect to your new Cassandra EC2 instance using any SSH client (PuTTY, Terminal, etc.). To connect, you will need a private key (the .pem file you created earlier) and the public DNS name of a node. Connect as user `ubuntu` rather than as `root`.

If this is the first time you are connecting, copy your private key file (<keyname>.pem) you downloaded earlier to your home directory, and change the permissions so it is not publicly viewable. For example:

```
chmod 400 datastax-key.pem
```

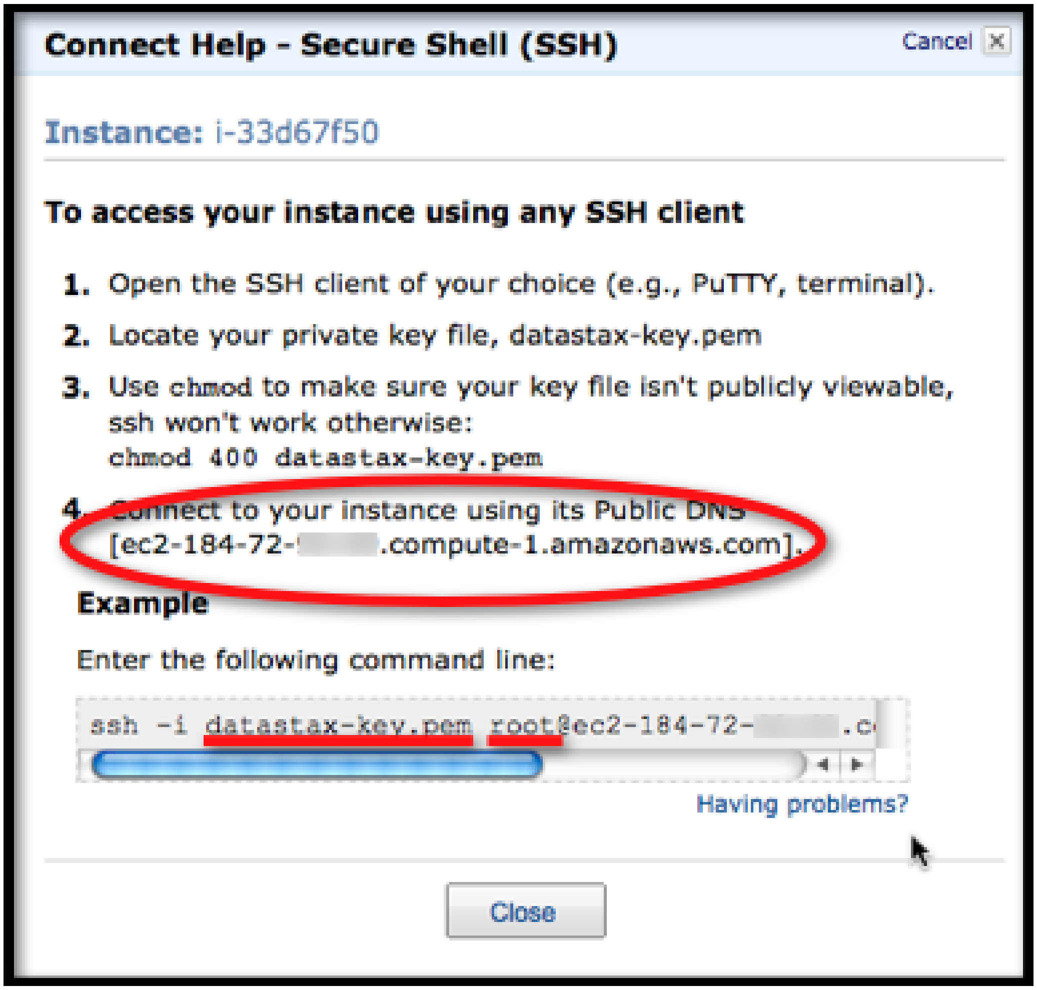
1. From the **My Instances** page in your AWS EC2 Dashboard, select the node you want to connect to. Since all nodes are peers in Cassandra, you can connect using any node in the cluster. However, the first node is typically the node running the OpsCenter service (and is also the Cassandra seed node).



	Instance	NS	Private DNS	Security Group IDs	Key Pair Name
☑	i-33d67f50	2-92	ip-10-111-29-7	sg-33f8e55a	datastax-key
☐	i-31d67f52	-123	ip-10-2-206-50	sg-33f8e55a	datastax-key
☐	i-37d67f54	-22-	domU-12-31-3	sg-33f8e55a	datastax-key
☐	i-35d67f56	-117	domU-12-31-3	sg-33f8e55a	datastax-key
☐	i-3bd67f58	2-73	ip-10-140-0-35	sg-33f8e55a	datastax-key
☐	i-39d67f5a	36-2	ip-10-46-215-1	sg-33f8e55a	datastax-key

2. To get the public DNS name of a node, select **Instance Actions > Connect**

3. This will open a **Connect Help - Secure Shell (SSH)** page for the selected node. This page will have all of the information you need to connect via SSH. If you copy and paste the command line, change the connection user from `root` to `ubuntu`.



Connect Help - Secure Shell (SSH)

Instance: i-33d67f50

To access your instance using any SSH client

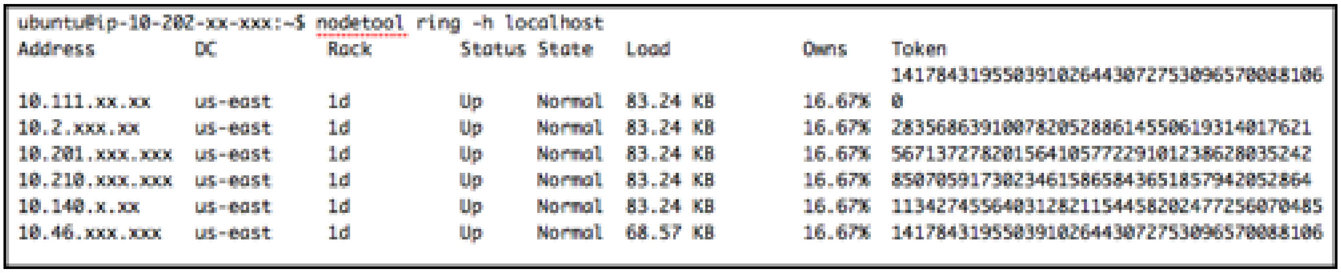
1. Open the SSH client of your choice (e.g., PuTTY, terminal).
2. Locate your private key file, datastax-key.pem
3. Use chmod to make sure your key file isn't publicly viewable, ssh won't work otherwise:
   chmod 400 datastax-key.pem
4. Connect to your instance using its Public DNS [ec2-184-72-[redacted].compute-1.amazonaws.com].

Example

Enter the following command line:

```
ssh -i datastax-key.pem root@ec2-184-72-[redacted].c
```

4. The AMI image configures your cluster and starts the Cassandra and OpsCenter services. After you have logged in to a node, run the *nodetool ring* command to make sure your cluster is up and running. For example:



```
ubuntu@ip-10-202-xx-xxx:~$ nodetool ring -h localhost
Address          DC          Rack        Status State   Load            Owns    Token
                                                                                141784319550391026443072753096570088106
10.111.xx.xx     us-east     1d          Up     Normal  83.24 KB        16.67%  0
10.2.xxx.xx      us-east     1d          Up     Normal  83.24 KB        16.67%  28356863910078205288614550619314017621
10.201.xxx.xxx   us-east     1d          Up     Normal  83.24 KB        16.67%  56713727820156410577229101238628035242
10.210.xxx.xxx   us-east     1d          Up     Normal  83.24 KB        16.67%  85070591730234615865843651857942052864
10.140.x.xx      us-east     1d          Up     Normal  83.24 KB        16.67%  113427455640312821154458202477256070485
10.46.xxx.xxx    us-east     1d          Up     Normal  68.57 KB        16.67%  141784319550391026443072753096570088106
```

5. For next steps, see *Running the Portfolio Demo Sample Application*.

Note

If you are installing OpsCenter with your Cassandra cluster, allow about 60 to 90 seconds after the cluster has finished loading for OpsCenter to start. You can launch OpsCenter using the URL: `http://<public-dns-of-first-instance>:8888`. After OpsCenter loads, you must install the OpsCenter agents if you want to see cluster performance data (click **Install Agents**). When prompted for credentials for the agent nodes, use the username `ubuntu` and copy/paste the entire contents from your private key file (the `.pem` file you downloaded earlier).

Configuring and Starting a Cassandra Cluster

The process for initializing a Cassandra cluster (be it a single node, multiple node, or multiple data center cluster) is to first correctly configure the *Node and Cluster Initialization Properties* in each node's *cassandra.yaml* configuration file, and then start each node individually starting with the seed node(s).

For more guidance on choosing the right configuration properties for your needs, see *Choosing Node Configuration Options*.

Initializing a Multi-Node or Multi-Data Center Cluster

To correctly configure a multi-node or multi-data center cluster you must determine the following information:

- A name for your cluster.
- How many total nodes your cluster will have, and how many nodes per data center (or replication group).
- The IP addresses of each node
- The token for each node (see *Calculating Tokens*). If you are deploying a multi-data center cluster, make sure to assign tokens so that data is evenly distributed within each data center or replication grouping (see *Calculating Tokens for a Multi-Data Center Cluster*).
- Which nodes will serve as the seed nodes. If you are deploying a multi data center cluster, the seed list should include a node from *each* data center or replication group.
- The *snitch* you plan to use.

This information will be used to configure the *Node and Cluster Initialization Properties* in the *cassandra.yaml* configuration file on each node in the cluster. Each node should be correctly configured before starting up the cluster, one node at a time (starting with the seed nodes).

For example, suppose you are configuring a 6 node cluster spanning 2 racks in a single data center. The nodes have the following IPs, and one node per rack will serve as a seed:

- node0 110.82.155.0 (seed1)
- node1 110.82.155.1
- node2 110.82.155.2
- node3 110.82.156.3 (seed2)
- node4 110.82.156.4
- node5 110.82.156.5

The *cassandra.yaml* files for each node would then have the following modified property settings.

node0

```
cluster_name: 'MyDemoCluster'
initial_token: 0
seed_provider:
        - seeds: "110.82.155.0,110.82.155.3"
```

```
listen_address: 110.82.155.0
rpc_address: 0.0.0.0
endpoint_snitch: RackInferringSnitch
```

node1

```
cluster_name: 'MyDemoCluster'
initial_token: 28356863910078205288614550619314017621
seed_provider:
       - seeds: "110.82.155.0,110.82.155.3"
listen_address: 110.82.155.1
rpc_address: 0.0.0.0
endpoint_snitch: RackInferringSnitch
```

node2

```
cluster_name: 'MyDemoCluster'
initial_token: 56713727820156410577229101238628035242
seed_provider:
       - seeds: "110.82.155.0,110.82.155.3"
listen_address: 110.82.155.2
rpc_address: 0.0.0.0
endpoint_snitch: RackInferringSnitch
```

node3

```
cluster_name: 'MyDemoCluster'
initial_token: 85070591730234615865843651857942052864
seed_provider:
       - seeds: "110.82.155.0,110.82.155.3"
listen_address: 110.82.155.3
rpc_address: 0.0.0.0
endpoint_snitch: RackInferringSnitch
```

node4

```
cluster_name: 'MyDemoCluster'
initial_token: 113427455640312821154458202477256070485
seed_provider:
       - seeds: "110.82.155.0,110.82.155.3"
listen_address: 110.82.155.4
rpc_address: 0.0.0.0
endpoint_snitch: RackInferringSnitch
```

node5

```
cluster_name: 'MyDemoCluster'
initial_token: 141784319550391026443072753096570088106
seed_provider:
       - seeds: "110.82.155.0,110.82.155.3"
listen_address: 110.82.155.5
rpc_address: 0.0.0.0
endpoint_snitch: RackInferringSnitch
```

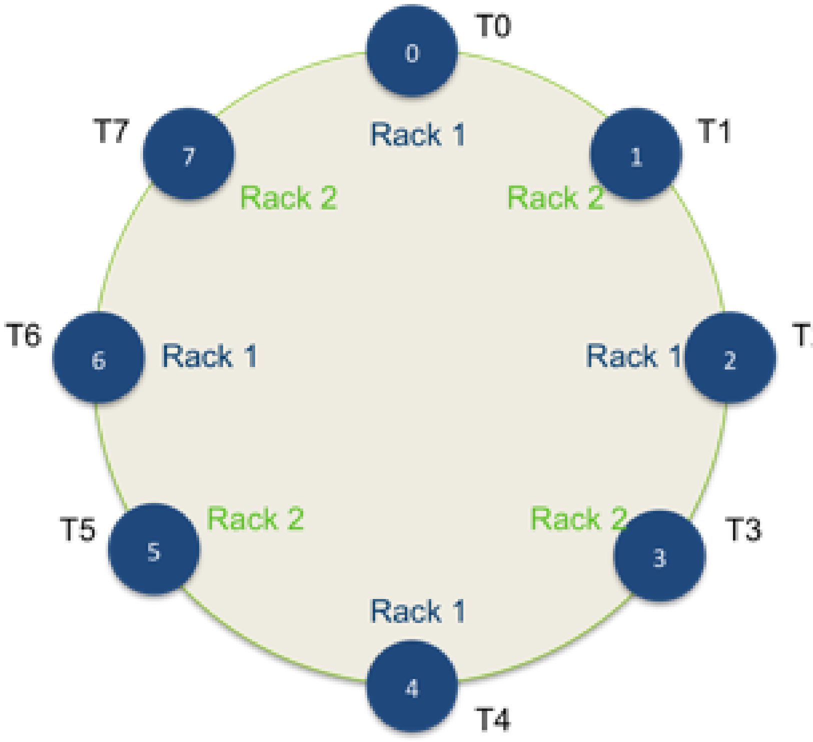
Calculating Tokens

Tokens are used to assign a range of data to a particular node. Assuming you are using the `RandomPartitioner` (the default partitioner), the approaches described in this section will ensure even data distribution.

Each node in the cluster should be assigned a token before it is started for the first time. The token is set with the *initial_token* property in the *cassandra.yaml* configuration file.

Calculating Tokens for Multiple Racks

If you have multiple racks in single data center or a multiple data center cluster, you can use the same formula for calculating the tokens. However you should assign the tokens to nodes in alternating racks. For example: rack1, rack2, rack3, rack1, rack2, rack3, and so on. Be sure to have the same number of nodes in each rack.



Assign tokens to nodes in alternating racks

Calculating Tokens for a Single Data Center

1. Create a new file for your token generator program:

```
vi tokengentool
```

2. Paste the following Python program into this file:

```
#! /usr/bin/python
import sys
if (len(sys.argv) > 1):
    num=int(sys.argv[1])
else:
    num=int(raw_input("How many nodes are in your cluster? "))
for i in range(0, num):
    print 'token %d: %d' % (i, (i*(2**127)/num))
```

3. Save and close the file and make it executable:

```
chmod +x tokengentool
```

4. Run the script:

```
./tokengentool
```

5. When prompted, enter the total number of nodes in your cluster:

```
How many nodes are in your cluster? 6
token 0: 0
token 1: 28356863910078205288614550619314017621
token 2: 56713727820156410577229101238628035242
token 3: 85070591730234615865843651857942052864
token 4: 113427455640312821154458202477256070485
token 5: 141784319550391026443072753096570088106
```

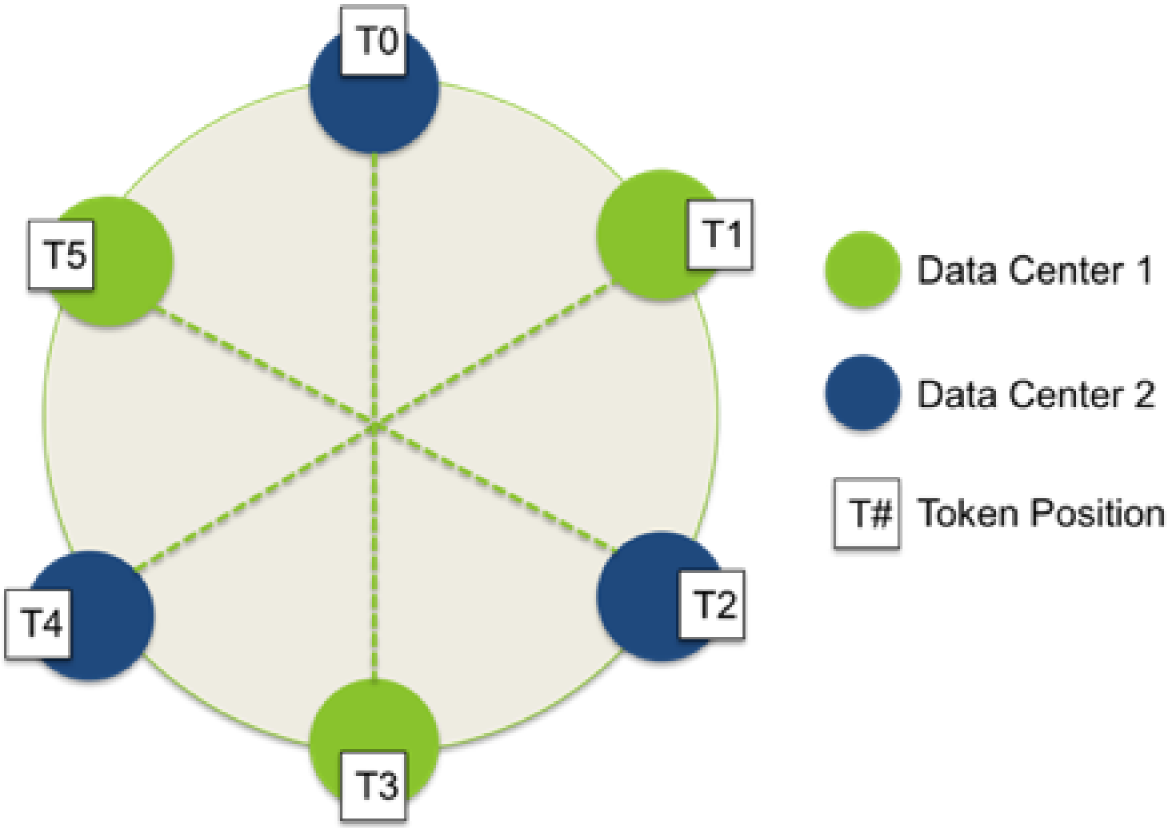
6. On each node, edit the `cassandra.yaml` file and enter its corresponding token value in the `initial_token` property.

Calculating Tokens for a Multi-Data Center Cluster

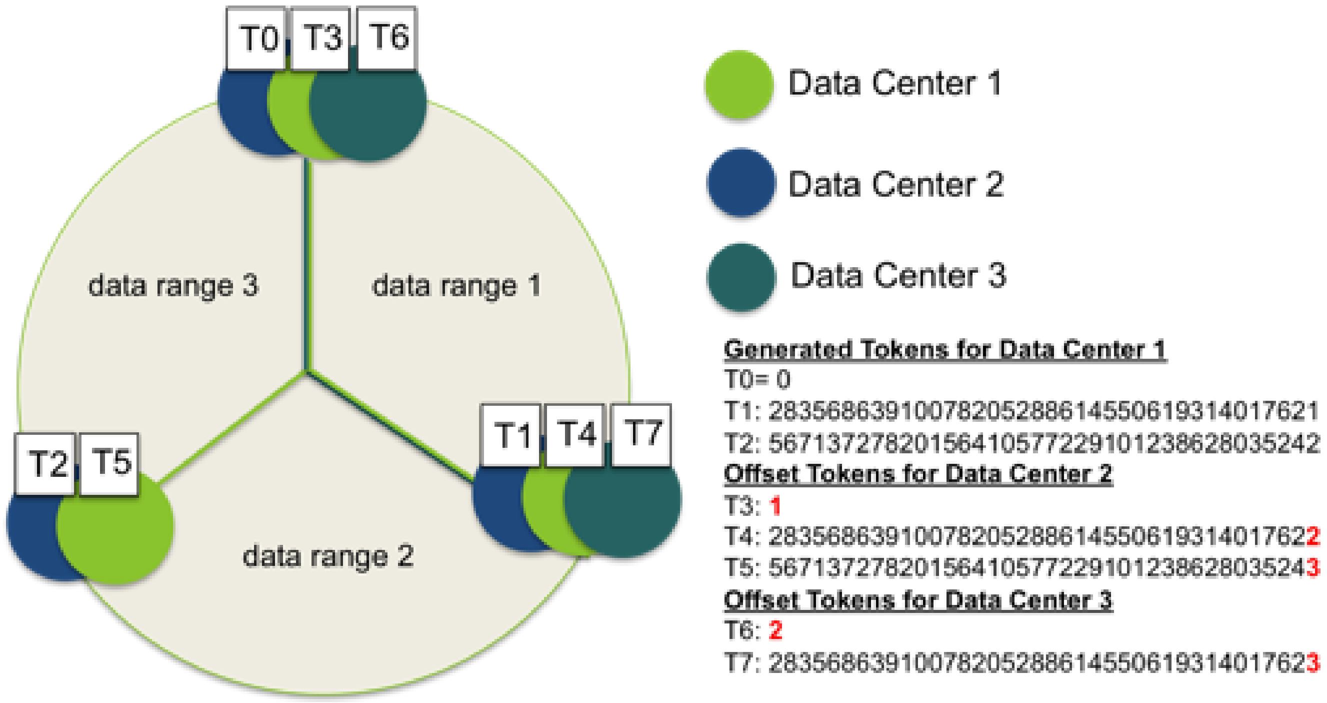
In multi-data center deployments, replica placement is calculated per data center using the `NetworkTopologyStrategy` replica placement strategy. In each data center (or replication group) the first replica for a particular row is determined by the token value assigned to a node. Additional replicas in the same data center are placed by walking the ring clockwise until it reaches the first node in another rack.

If you do not calculate partitioner tokens so that the data ranges are evenly distributed for each data center, you could end up with uneven data distribution within a data center. The goal is to ensure that the nodes for each data center are evenly dispersed around the ring, or to calculate tokens for each replication group individually (without conflicting token assignments).

One way to avoid uneven distribution is to calculate tokens for all nodes in the cluster, and then alternate the token assignments so that the nodes for each data center are evenly dispersed around the ring.



Another way to assign tokens in a multi data center cluster is to generate tokens for the nodes in one data center, and then offset those token numbers by 1 for all nodes in the next data center, by 2 for the nodes in the next data center, and so on. This approach is good if you are adding a data center to an established cluster, or if your data centers do not have the same number of nodes.



Generated Tokens for Data Center 1
T0= 0
T1: 28356863910078205288614550619314017621
T2: 56713727820156410577229101238628035242
Offset Tokens for Data Center 2
T3: 1
T4: 28356863910078205288614550619314017622
T5: 56713727820156410577229101238628035243
Offset Tokens for Data Center 3
T6: 2
T7: 28356863910078205288614550619314017623

Starting and Stopping a Cassandra Node

After you have installed and configured Cassandra on all nodes, you are ready to start your cluster. On initial start-up, each node must be started one at a time, starting with your seed nodes.

Packaged installations include startup scripts for running Cassandra as a service. Binary packages do not.

- *Starting/Stopping Cassandra as a Stand-Alone Process*
- *Starting/Stopping Cassandra as a Service*

Starting/Stopping Cassandra as a Stand-Alone Process

You can start the Cassandra Java server process as follows:

```
$ cd $CASSANDRA_HOME
$ sh bin/cassandra -f
```

To stop the Cassandra process, find the Cassandra Java process ID (PID), and then `kill -9` the process using its PID number. For example:

```
$ ps ax | grep java
$ kill -9 1539
```

Starting/Stopping Cassandra as a Service

Packaged installations provide startup scripts in `/etc/init.d` for starting Cassandra as a service. The service runs as the `cassandra` user. You must have root or sudo permissions to start or stop services.

To start the Cassandra service (as root):

```
# service cassandra start
```

To stop the Cassandra service (as root):

```
# service cassandra stop
```

Note

On Enterprise Linux systems, the Cassandra service runs as a `java` process. On Debian systems, the Cassandra service runs as a `jsvc` process.

Upgrading Cassandra

This section includes information on upgrading between releases:

Major Releases	Minor Releases
Upgrading Cassandra: 0.8.x to 1.0.x	Upgrading Between Minor Releases of Cassandra 1.0.x

Best Practices for Upgrading Cassandra

The following best practices are recommended when upgrading Cassandra:

- Always take a snapshot before any upgrade. This allows you to rollback to the previous version if necessary. Cassandra is able to read data files created by the previous version, but the inverse is not always true.

  #### Note

  Snapshotting is fast, especially if you have JNA installed, and takes effectively zero disk space until you start compacting the live data files again.

- Be sure to check https://github.com/apache/cassandra/blob/trunk/NEWS.txt for any new information on upgrading.
- For a list of fixes and new features, see https://github.com/apache/cassandra/blob/trunk/CHANGES.txt

Upgrading Cassandra: 0.8.x to 1.0.x

Upgrading from version 0.8 or later can be done with a rolling restart, one node at a time. You do not need to bring down the whole cluster at once.

To upgrade a binary installation from 0.8.x to 1.0.x:

1. On each node, download and unpack the 1.0 binary tarball package from the downloads section of the Cassandra website.
2. Account for *New and Changed Parameters between 0.8 and 1.0* in `cassandra.yaml`. You can copy your existing 0.8.x configuration file into the upgraded Cassandra instance and manually update it with new content.
3. Make sure any client drivers -- such as Hector or Pycassa clients -- are 1.0-compatible.
4. Run `nodetool drain` on the 0.8.x node to flush the commit log.
5. Stop the old Cassandra process, then start the new binary process.
6. Monitoring the log files for any issues.
7. After upgrading and restarting all Cassandra processes, restart client applications.
8. After upgrading, run *nodetool upgradesstables* against each node before running repair, moving nodes, or adding new ones. (If using Cassandra 1.0.3 and earlier, use *nodetool scrub* instead.)

To upgrade a CentOS/RHEL packaged release installation from 0.8.x to 1.0.x:

1. On each of your Cassandra nodes, run `sudo yum install apache-cassandra1`.
2. Account for *New and Changed Parameters between 0.8 and 1.0* in `cassandra.yaml`. The installer creates the file `cassandra.yaml.rpmnew` in `/etc/cassandra/default.conf/`. You can diff this file with your existing configuration and add new content.
3. Make sure any client drivers -- such as Hector or Pycassa clients -- are 1.0-compatible.
4. Run `nodetool drain` on the 0.8.x node to flush the commit log.
5. Restart the Cassandra process.
6. Monitor the log files for any issues.
7. After upgrading and restarting all Cassandra processes, restart client applications.
8. After upgrading, run *nodetool upgradesstables* against each node before running repair, moving nodes, or adding new ones. (If using Cassandra 1.0.3 and earlier, use *nodetool scrub* instead.)

To upgrade a Debian/Ubuntu packaged release installation from 0.8.x to 1.0.x:

1. On each of your Cassandra nodes, run `sudo apt-get install cassandra1`.
2. Account for *New and Changed Parameters between 0.8 and 1.0* in `cassandra.yaml`. The installer creates the file `cassandra.yaml.rpmnew` in `/etc/cassandra/default.conf/`. You can diff this file with your existing configuration and add new content.
3. Make sure any client drivers, such as Hector or Pycassa clients, are 1.0-compatible.
4. Run `nodetool drain` on the 0.8.x node to flush the commit log.
5. Restart the Cassandra process.
6. Monitor the log files for any issues.
7. After upgrading and restarting all Cassandra processes, restart client applications.
8. After upgrading, run *nodetool upgradesstables* against each node before running repair, moving nodes, or adding new ones. (If using Cassandra 1.0.3 and earlier, use *nodetool scrub* instead.)

New and Changed Parameters between 0.8 and 1.0

This table lists `cassandra.yaml` parameters that have changed between 0.8 and 1.0. See the *cassandra.yaml reference* for details on these parameters.

Option	Default Value
1.0 Release	
broadcast_address	Same as listen_address - set to the public IP in multi-region EC2 clusters
compaction_thread_priority	Removed (use *compaction_throughput_mb_per_sec* instead)
commitlog_rotation_threshold_in_mb	Removed
commitlog_total_space_in_mb	4096 (replaces column family storage property *memtable_flush_after_mins*)
multithreaded_compaction	false
memtable_total_space_in_mb	1/3 of heap (replaces column family storage properties *memtable_operations_in_millions* and *memtable_throughput_in_mb*)
0.8 Release	
seed_provider	SimpleSeedProvider
seeds	Note: Now a comma-delimited list in double quotes.

memtable_total_space_in_mb	1/3 of heap
compaction_throughput_mb_per_sec	16
concurrent_compactors	One per CPU
internode_encryption	none
keystore	conf/.keystore
keystore_password	cassandra
truststore	conf/.truststore
truststore_password	cassandra

Upgrading Between Minor Releases of Cassandra 1.0.x

Upgrading minor releases can be done with a rolling restart, one node at a time. You do not need to bring down the whole cluster at once.

To upgrade a binary tarball package installation:

1. On each node, download and unpack the binary tarball package from the downloads section of the Cassandra website.
2. Account for *New and Changed Parameters between 0.8 and 1.0* in `cassandra.yaml`. You can copy your existing configuration file into the upgraded Cassandra instance and manually update it with new content.
3. Make sure any client drivers, such as Hector or Pycassa clients, are compatible with the new version.
4. Flush the commit log on the upgraded node by running `nodetool drain`.
5. Stop the old Cassandra process, then start the new binary process.
6. Monitor the log files for any issues.

To upgrade a Debian or RPM package installation:

1. On each node, download and install the package from the downloads section of the Cassandra website.
2. Account for *New and Changed Parameters between 0.8 and 1.0* in `cassandra.yaml`. You can copy your existing configuration file into the upgraded Cassandra instance and manually update it with new content.
3. Make sure any client drivers, such as Hector or Pycassa clients, are compatible with the new version.
4. Flush the commit log on the upgraded node by running `nodetool drain`.
5. Restart the Cassandra process.
6. Monitor the log files for any issues.

Understanding the Cassandra Data Model

The Cassandra data model is a dynamic schema, column-oriented data model. This means that, unlike a relational database, you do not need to model all of the columns required by your application up front, as each row is not required to have the same set of columns. Columns and their metadata can be added by your application as they are needed without incurring downtime to your application.

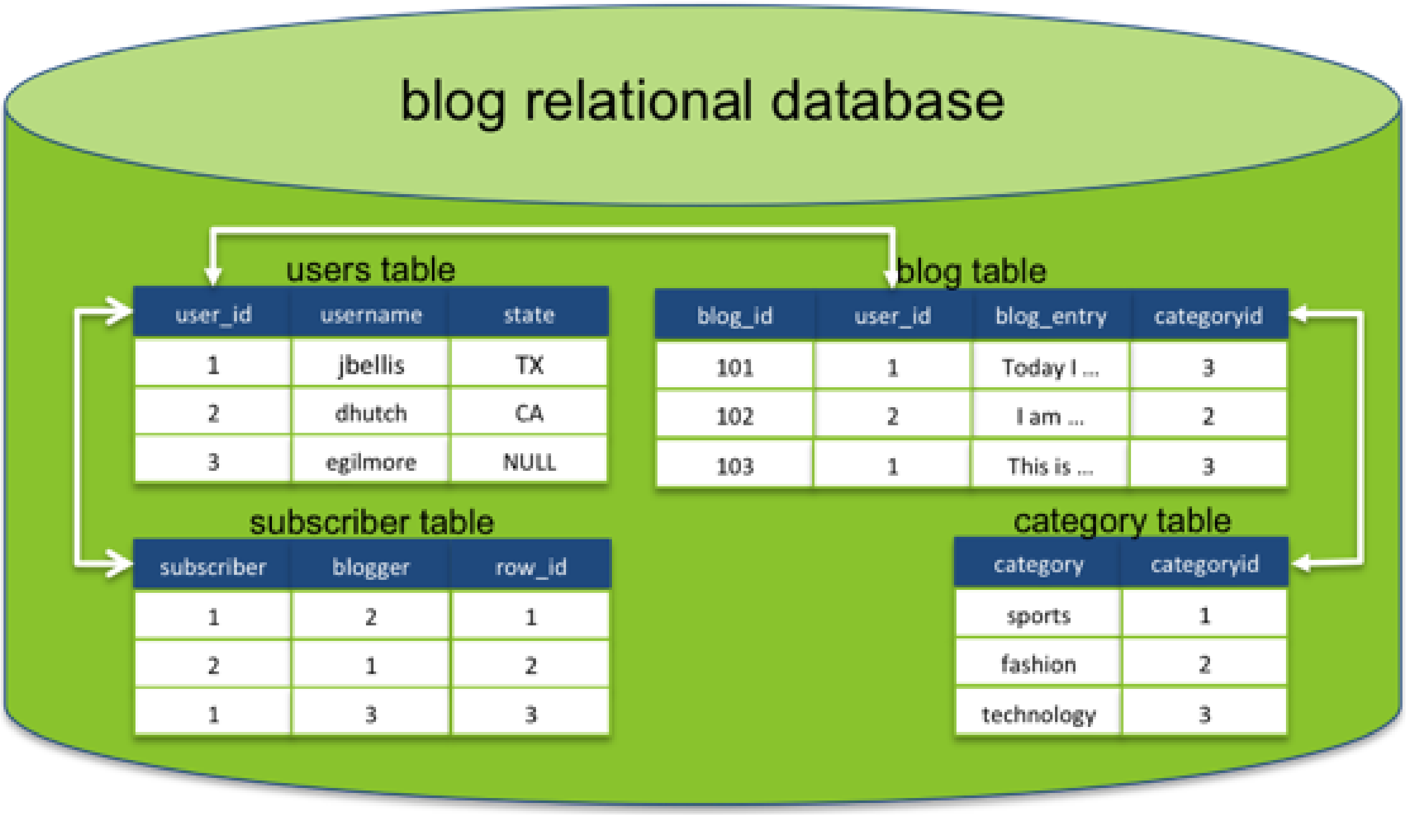
The Cassandra Data Model

For developers new to Cassandra and coming from a relational database background, the data model can be a bit confusing. The following section provides a comparison of the two.

Comparing the Cassandra Data Model to a Relational Database

The Cassandra data model is designed for distributed data on a very large scale. Although it is natural to want to compare the Cassandra data model to a relational database, they are really quite different. In a relational database, data is stored in tables and the tables comprising an application are typically related to each other. Data is usually normalized to reduce redundant entries, and tables are joined on common keys to satisfy a given query.

For example, consider a simple application that allows users to create blog entries. In this application, blog entries are categorized by subject area (sports, fashion, etc.). Users can also choose to subscribe to the blogs of other users. In this example, the user id is the primary key in the *users* table and the foreign key in the *blog* and *subscriber* tables. Likewise, the category id is the primary key of the *category* table and the foreign key in the *blog_entry* table. Using this relational model, SQL queries can perform joins on the various tables to answer questions such as "what users subscribe to my blog" or "show me all of the blog entries about fashion" or "show me the most recent entries for the blogs I subscribe to".



users table

user_id	username	state
1	jbellis	TX
2	dhutch	CA
3	egilmore	NULL

blog table

blog_id	user_id	blog_entry	categoryid
101	1	Today I ...	3
102	2	I am ...	2
103	1	This is ...	3

subscriber table

subscriber	blogger	row_id
1	2	1
2	1	2
1	3	3

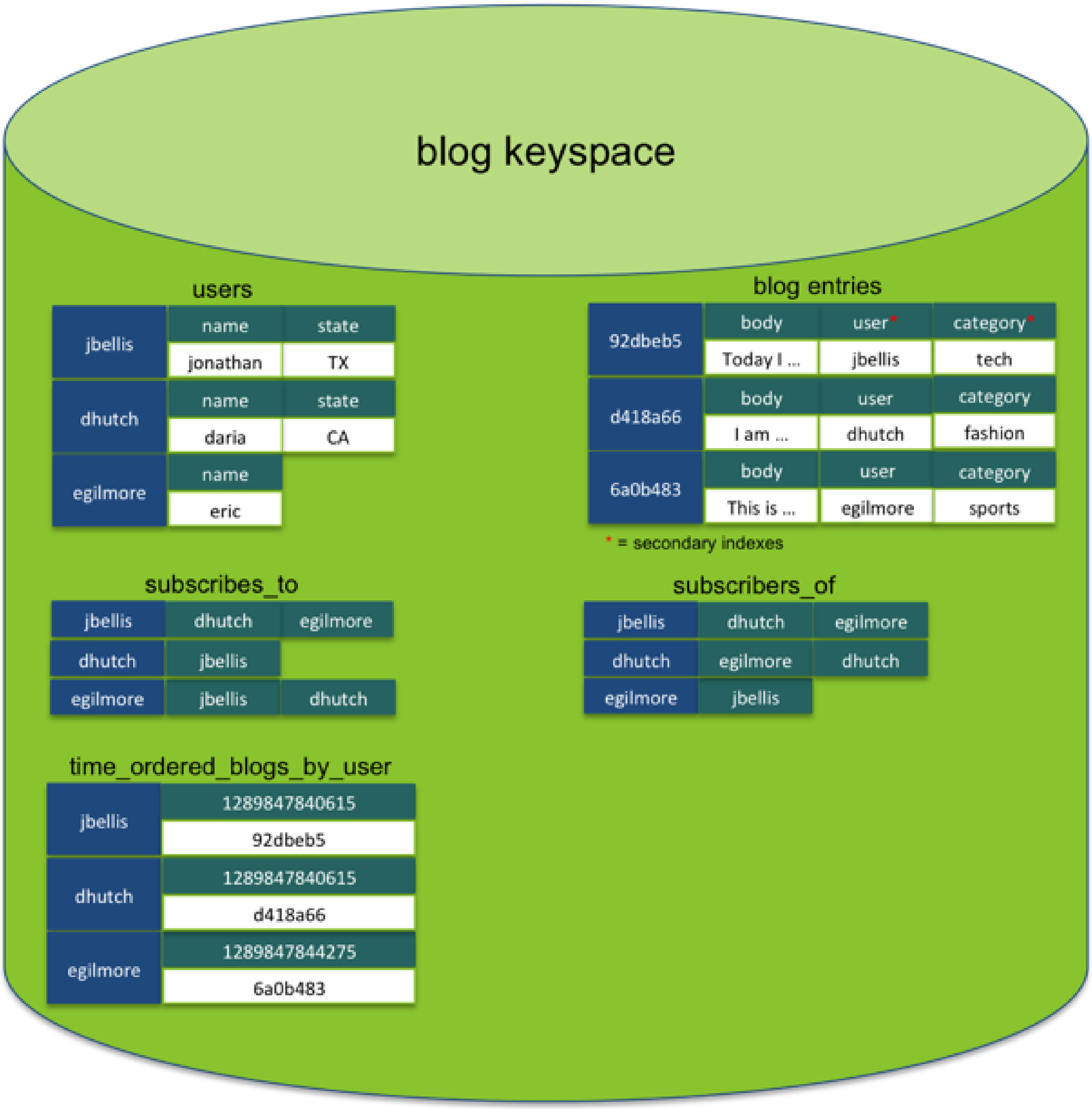
category table

category	categoryid
sports	1
fashion	2
technology	3

In Cassandra, the *keyspace* is the container for your application data, similar to a database or schema in a relational database. Inside the keyspace are one or more *column family* objects, which are analogous to tables. Column families contain *columns*, and a set of related columns is identified by an application-supplied row *key*. Each row in a column family is *not* required to have the same set of columns.

Cassandra does not enforce relationships between column families the way that relational databases do between tables: there are no formal foreign keys in Cassandra, and joining column families at query time is not supported. Each column family has a self-contained set of columns that are intended to be accessed together to satisfy specific queries from your application.

For example, using the blog application example, you might have a column family for user data and blog entries similar to the relational model. Other column families (or secondary indexes) could then be added to support the queries your application needs to perform. For example, to answer the queries "what users subscribe to my blog" or "show me all of the blog entries about fashion" or "show me the most recent entries for the blogs I subscribe to", you would need to design additional column families (or add secondary indexes) to support those queries. Keep in mind that some denormalization of data is usually required.



users

jbellis	name	state
	jonathan	TX
dhutch	name	state
	daria	CA
egilmore	name	
	eric	

blog entries

92dbeb5	body	user*	category*
	Today I ...	jbellis	tech
d418a66	body	user	category
	I am ...	dhutch	fashion
6a0b483	body	user	category
	This is ...	egilmore	sports

* = secondary indexes

subscribes_to

jbellis	dhutch	egilmore
dhutch	jbellis	
egilmore	jbellis	dhutch

subscribers_of

jbellis	dhutch	egilmore
dhutch	egilmore	dhutch
egilmore	jbellis	

time_ordered_blogs_by_user

jbellis	1289847840615
	92dbeb5
dhutch	1289847840615
	d418a66
egilmore	1289847844275
	6a0b483

About Keyspaces

In Cassandra, the *keyspace* is the container for your application data, similar to a schema in a relational database. Keyspaces are used to group column families together. Typically, a cluster has one keyspace per application.

Replication is controlled on a per-keyspace basis, so data that has different replication requirements should reside in different keyspaces. Keyspaces are not designed to be used as a significant map layer within the data model, only as a way to control data replication for a set of column families.

Defining Keyspaces

Data Definition Language (DDL) commands for defining and altering keyspaces are provided in the various client interfaces, such as Cassandra CLI and CQL. For example, to define a keyspace in CQL:

```
CREATE KEYSPACE keyspace_name WITH
strategy_class = 'SimpleStrategy'
AND strategy_options:replication_factor=2;
```

Or in Cassandra CLI:

```
CREATE KEYSPACE keyspace_name WITH
placement_strategy = 'SimpleStrategy'
AND strategy_options = [{replication_factor:2}];
```

See *Getting Started Using the Cassandra CLI* and *Getting Started with CQL* for more information on DDL commands for Cassandra.

About Column Families

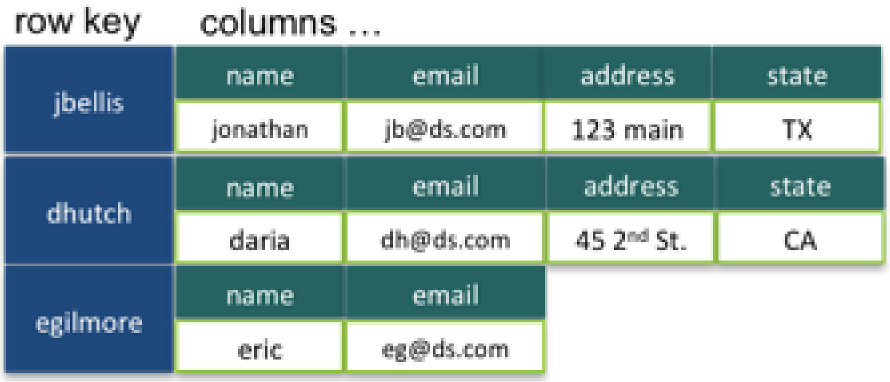
When comparing Cassandra to a relational database, the column family is similar to a table in that it is a container for columns and rows. However, a column family requires a major shift in thinking for those coming from the relational world.

In a relational database, you define tables, which have defined columns. The table defines the column names and their data types, and the client application then supplies rows conforming to that schema: each row contains the same fixed set of columns.

In Cassandra, you define column families. Column families can (and should) define metadata about the columns, but the actual columns that make up a row are determined by the client application. Each row can have a different set of columns.

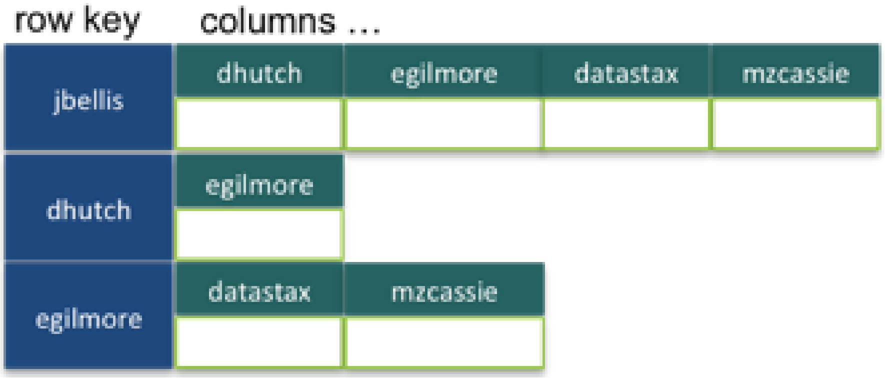
Although column families are very flexible, in practice a column family is not entirely schema-less. Each column family should be designed to contain a single type of data. There are two typical column family design patterns in Cassandra; the *static* and *dynamic* column families.

A static column family uses a relatively static set of column names and is more similar to a relational database table. For example, a column family storing user data might have columns for the user name, address, email, phone number and so on. Although the rows will generally have the same set of columns, they are not required to have all of the columns defined. Static column families typically have column metadata pre-defined for each column.



row key	columns ...			
jbellis	name	email	address	state
	jonathan	jb@ds.com	123 main	TX
dhutch	name	email	address	state
	daria	dh@ds.com	45 2^{nd} St.	CA
egilmore	name	email		
	eric	eg@ds.com		

A dynamic column family takes advantage of Cassandra's ability to use arbitrary application-supplied column names to store data. A dynamic column family allows you to pre-compute result sets and store them in a single row for efficient data retrieval. Each row is a snapshot of data meant to satisfy a given query, sort of like a materialized view. For example, a column family that tracks the users that subscribe to a particular user's blog.

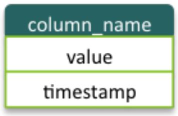


Instead of defining metadata for individual columns, a dynamic column family defines the type information for column names and values (comparators and validators), but the actual column names and values are set by the application when a column is inserted.

For all column families, each row is uniquely identified by its row *key*, similar to the primary key in a relational table. A column family is always partitioned on its row key, and the row key is always implicitly indexed.

About Columns

The column is the smallest increment of data in Cassandra. It is a tuple containing a name, a value and a timestamp.



A column must have a name, and the name can be a static label (such as "name" or "email") or it can be dynamically set when the column is created by your application.

Columns can be indexed on their name (see *secondary indexes*). However, one limitation of column indexes is that they do not support queries that require access to ordered data, such as time series data. In this case a secondary index on a timestamp column would not be sufficient because you cannot control column sort order with a secondary index. For cases where sort order is important, manually maintaining a column family as an 'index' is another way to lookup column data in sorted order.

It is not required for a column to have a value. Sometimes all the information your application needs to satisfy a given query can be stored in the column name itself. For example, if you are using a column family as a materialized view to lookup rows from other column families, all you need to store is the row key that you are looking up; the value can be empty.

Cassandra uses the column timestamp to determine the most recent update to a column. The timestamp is provided by the client application. The latest timestamp always wins when requesting data, so if multiple client sessions update the same columns in a row concurrently, the most recent update is the one that will eventually persist. See *About Transactions and Concurrency Control* for more information about how Cassandra handles conflict resolution.

About Special Columns (Counter, Expiring, Super)

Cassandra has three special types of columns, described below:

About Expiring Columns

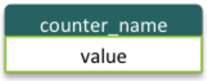
A column can also have an optional expiration date, known in Cassandra as the *time to live* (TTL). Whenever a column is inserted, the client request can specify an optional TTL value for the column. TTL columns are marked as deleted (with a tombstone) after the requested amount of time has expired. Once they are marked as deleted, they are automatically removed during the normal compaction and repair processes.

About Counter Columns

A counter is a special kind of column used to store a number that incrementally counts the occurrences of a particular event or process. For example, you might use a counter column to count the number of times a page is viewed.

Counter column families must use CounterColumnType as the validator (the column value type). This means that currently, counters may only be stored in dedicated column families; they will be allowed to mix with normal columns in a future release.

Counter columns are different from regular columns in that once a counter is defined, the client application then updates the column value by incrementing (or decrementing) it. A client update to a counter column passes the name of the counter and the increment (or decrement) value; no timestamp is required.

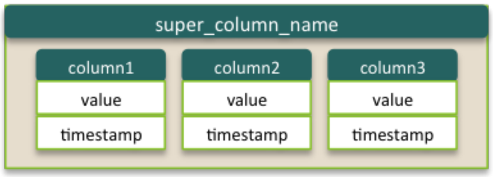


Internally, the structure of a counter column is a bit more complex. Cassandra tracks the distributed state of the counter as well as a server-generated timestamp upon deletion of a counter column. For this reason, it is important that all nodes in your cluster have their clocks synchronized using network time protocol (NTP).

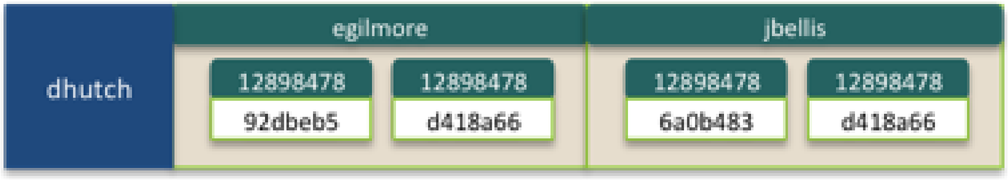
A counter can be read or written at any of the available *consistency levels*. However, it's important to understand that unlike normal columns, a write to a counter requires a read in the background to ensure that distributed counter values remain consistent across replicas. If you write at a consistency level of ONE, the implicit read will not impact write latency, hence, ONE is the most common consistency level to use with counters.

About Super Columns

A Cassandra column family can contain either regular columns or *super columns*, which adds another level of nesting to the regular column family structure. Super columns are comprised of a (super) column name and an ordered map of sub-columns. A super column can specify a comparator on both the super column name as well as on the sub-column names.



A super column is a way to group multiple columns based on a common lookup value. The primary use case for super columns is to denormalize multiple rows from other column families into a single row, allowing for materialized view data retrieval. For example, suppose you wanted to create a materialized view of blog entries for the bloggers that a user follows.



One limitation of super columns is that all sub-columns of a super column must be deserialized in order to read a single sub-column value, and you cannot create secondary indexes on the sub-columns of a super column. Therefore, the use of super columns is best suited for use cases where the number of sub-columns is a relatively small number.

About Data Types (Comparators and Validators)

In a relational database, you must specify a data type for each column when you define a table. The data type constrains the values that can be inserted into that column. For example, if you have a column defined as an integer datatype, you would not be allowed to insert character data into that column. Column names in a relational database are typically fixed labels (strings) that are assigned when you define the table schema.

In Cassandra, the data type for a column (or row key) *value* is called a *validator*. The data type for a column *name* is called a *comparator*. You can define data types when you create your column family schemas (which is recommended), but Cassandra does not require it. Internally, Cassandra stores column names and values as hex byte arrays (`BytesType`). This is the default client encoding used if data types are not defined in the column family schema (or if not specified by the client request).

Cassandra comes with the following built-in data types, which can be used as both validators (row key and column value data types) or comparators (column name data types). One exception is `CounterColumnType`, which is only allowed as a column value (not allowed for row keys or column names).

Internal Type	CQL Name	Description
BytesType	blob	Arbitrary hexadecimal bytes (no validation)
AsciiType	ascii	US-ASCII character string
UTF8Type	text, varchar	UTF-8 encoded string
IntegerType	varint	Arbitrary-precision integer
LongType	int, bigint	8-byte long
UUIDType	uuid	Type 1 or type 4 UUID
DateType	timestamp	Date plus time, encoded as 8 bytes since epoch
BooleanType	boolean	true or false
FloatType	float	4-byte floating point
DoubleType	double	8-byte floating point
DecimalType	decimal	Variable-precision decimal
CounterColumnType	counter	Distributed counter value (8-byte long)

About Validators

For all column families, it is best practice to define a default row key validator using the *key_validation_class* property.

For static column families, you should define each column and its associated type when you define the column family using the *column_metadata* property.

For dynamic column families (where column names are not known ahead of time), you should specify a *default_validation_class* instead of defining the per-column data types.

Key and column validators may be added or changed in a column family definition at any time. If you specify an invalid validator on your column family, client requests that respect that metadata will be confused, and data inserts or updates that do not conform to the specified validator will be rejected.

About Comparators

Within a row, columns are always stored in sorted order by their *column name*. The *comparator* specifies the data type for the column name, as well as the sort order in which columns are stored within a row. Unlike validators, the comparator may *not* be changed after the column family is defined, so this is an important consideration when defining a column family in Cassandra.

Typically, static column family names will be strings, and the sort order of columns is not important in that case. For dynamic column families, however, sort order is important. For example, in a column family that stores time series data (the column names are timestamps), having the data in sorted order is required for slicing result sets out of a row of columns.

About Column Family Compression

Data compression can be configured on a per-column family basis. Compression maximizes the storage capacity of your Cassandra nodes by reducing the volume of data on disk. In addition to the space-saving benefits, compression also reduces disk I/O, particularly for read-dominated workloads.

Besides reducing data size, compression typically improves both read and write performance. Cassandra is able to quickly find the location of rows in the SSTable index, and only decompresses the relevant row chunks. This means compression improves read performance not just by allowing a larger data set to fit in memory, but it also benefits workloads where the hot data set does not fit into memory.

Unlike in traditional databases, write performance is not negatively impacted by compression in Cassandra. Writes on compressed tables can in fact show up to a 10 percent performance improvement. In traditional relational databases, writes require overwrites to existing data files on disk. This means that the database has to locate the relevant pages on disk, decompress them, overwrite the relevant data, and then compress them again (an expensive operation in both CPU cycles and disk I/O).

Because Cassandra SSTable data files are immutable (they are not written to again after they have been flushed to disk), there is no recompression cycle necessary in order to process writes. SSTables are only compressed once, when they are written to disk.

Enabling compression can yield the following benefits, depending on the data characteristics of the column family:

- 2x-4x reduction in data size
- 25-35% performance improvement on reads
- 5-10% performance improvement on writes

When to Use Compression

Compression is best suited for column families where there are many rows, with each row having the same columns, or at least many columns in common. For example, a column family containing user data such as username, email, etc., would be a good candidate for compression. The more similar the data across rows, the greater the compression ratio will be, and the larger the gain in read performance.

Compression is not as good a fit for column families where each row has a different set of columns, or where there are just a few very wide rows. Dynamic column families such as this will not yield good compression ratios.

Configuring Compression on a Column Family

When you create or update a column family, you can choose to make it a compressed column family by setting the *compression_options* attributes.

You can enable compression when you create a new column family, or update an existing column family to add compression later on. When you add compression to an existing column family, existing SSTables on disk are not compressed immediately. Any new SSTables that are created will be compressed, and any existing SSTables will be compressed during the normal Cassandra compaction process. If necessary, you can force existing SSTables to be rewritten and compressed by using *nodetool upgradesstables* (Cassandra 1.0.4 or later) or *nodetool scrub*.

For example, to create a new column family with compression enabled using the Cassandra CLI, you would do the following:

```
[default@demo] CREATE COLUMN FAMILY users WITH key_validation_class=UTF8Type AND column_metadata = [
{column_name: name, validation_class: UTF8Type} {column_name: email, validation_class: UTF8Type} {column_name:
state, validation_class: UTF8Type} {column_name: gender, validation_class: UTF8Type} {column_name: birth_year,
validation_class: LongType} ] AND compression_options={sstable_compression:SnappyCompressor,
chunk_length_kb:64};
```

About Indexes in Cassandra

An index is a data structure that allows for fast, efficient lookup of data matching a given condition.

About Primary Indexes

In relational database design, a primary key is the unique key used to identify each row in a table. A primary key index, like any index, speeds up random access to data in the table. The primary key also ensures record uniqueness, and may also control the order in which records are physically clustered, or stored by the database.

In Cassandra, the primary index for a column family is the index of its row keys. Each node maintains this index for the data it manages.

Rows are assigned to nodes by the cluster-configured *partitioner* and the keyspace-configured *replica placement strategy*. The primary index in Cassandra allows looking up of rows by their row key. Since each node knows what ranges of keys each node manages, requested rows can be efficiently located by scanning the row indexes only on the relevant replicas.

With randomly partitioned row keys (the default in Cassandra), row keys are partitioned by their MD5 hash and cannot be scanned in order like traditional b-tree indexes. Using an ordered partitioner does allow for range queries over rows, but is not recommended because of the difficulty in maintaining even data distribution across nodes. See *About Data Partitioning in Cassandra* for more information.

About Secondary Indexes

Secondary indexes in Cassandra refer to indexes on column values (to distinguish them from the primary row key index for a column family). Cassandra supports secondary indexes of the type `KEYS` (similar to a hash index).

Secondary indexes allow for efficient querying by specific values using equality predicates (where column *x* = value *y*). Also, queries on indexed values can apply additional filters to the result set for values of other columns.

Cassandra's built-in secondary indexes are best for cases when many rows contain the indexed value. The more unique values that exist in a particular column, the more overhead you will have, on average, to query and maintain the index. For example, suppose you had a user table with a billion users and wanted to look up users by the state they lived in. Many users will share the same column value for state (such as CA, NY, TX, etc.). This would be a good candidate for a secondary index. On the other hand, if you wanted to look up users by their email address (a value that is typically unique for each user), it may be more efficient to manually maintain a dynamic column family as a form of an "index". Even for columns containing unique data, it is often fine performance-wise to use secondary indexes for convenience, as long as the query volume to the indexed column family is moderate and not under constant load.

Another advantage of secondary indexes is the operational ease of populating and maintaining the index. When you create a secondary index on an existing column, it indexes the existing data in the background. Client-maintained 'column families as indexes' must be created manually; for example, if the `state` column had been indexed by creating a column family such as `users_by_state`, your client application would have to populate the column family with data from the `users` column family.

Building and Using Secondary Indexes

You can specify the `KEYS` index type when creating a column definition, or you can add it later to index an existing column. Secondary indexes are built in the background automatically, without blocking reads or writes.

For example, in the Cassandra CLI, you can create a secondary index on a column when defining a column family (note the `index_type:KEYS` specification for the `state` and `birth_year` columns):

```
[default@demo] create column family users with comparator=UTF8Type
... and column_metadata=[{column_name: full_name, validation_class: UTF8Type},
... {column_name: email, validation_class: UTF8Type},
... {column_name: birth_year, validation_class: LongType, index_type: KEYS},
... {column_name: state, validation_class:  UTF8Type, index_type: KEYS}];
```

Or you can add an index to an existing column family:

```
[default@demo] update column family users with comparator=UTF8Type
... and column_metadata=[{column_name: full_name, validation_class: UTF8Type},
... {column_name: email, validation_class: UTF8Type},
... {column_name: birth_year, validation_class: LongType, index_type: KEYS},
... {column_name: state, validation_class:  UTF8Type, index_type: KEYS}];
```

Because of the secondary index created for `state`, its values can then be queried directly for users who live in a given state. For example:

```
[default@demo] get users where state = 'TX';
```

Planning Your Data Model

Planning a data model in Cassandra has different design considerations than one may be used to from relational databases. Ultimately, the data model you design depends on the data you want to capture and how you plan to access it. However, there are some common design considerations for Cassandra data model planning.

Start with Queries

The best way to approach data modeling for Cassandra is to start with your queries and work backwards from there. Think about the actions your application needs to perform, how you want to access the data, and then design column families to support those access patterns.

For example, start with listing the all of the use cases your application needs to support. Think about the data you want to capture and the lookups your application needs to do. Also note any ordering, filtering or grouping requirements. For example, if you need events in chronological order, or if you only care about the last 6 months worth of data, those would be factors in your data model design for Cassandra.

Denormalize to Optimize

In the relational world, the data model is usually designed up front with the goal of normalizing the data to minimize redundancy. Normalization typically involves creating smaller, well-structured tables and then defining relationships between them. During queries, related tables are joined to satisfy the request.

Cassandra does not have foreign key relationships like a relational database does, which means you cannot join multiple column families to satisfy a given query request. Cassandra performs best when the data needed to satisfy a given query is located in the *same* column family. Try to plan your data model so that one or more rows in a single column family are used to answer each query. This sacrifices disk space (one of the cheapest resources for a server) in order to reduce the number of disk seeks and the amount of network traffic.

Planning for Concurrent Writes

Within a column family, every row is known by its row key, a string of virtually unbounded length. The key has no required form, but it must be unique within a column family. Unlike the primary key in a relational database, Cassandra does not enforce unique-ness. Inserting a duplicate row key will *upsert* the columns contained in the insert statement rather than return a unique constraint violation.

Using Natural or Surrogate Row Keys

One consideration is whether to use surrogate or natural keys for a column family. A surrogate key is a generated key (such as a UUID) that uniquely identifies a row, but has no relation to the actual data in the row.

For some column families, the data may contain values that are guaranteed to be unique and are not typically updated after a row is created. For example, the username in a users column family. This is called a natural key. Natural keys make the data more readable, and remove the need for additional indexes or denormalization. However, unless your client application ensures unique-ness, there is potential of over-writing column data.

Also, the natural key approach does not easily allow updates to the row key. For example, if your row key was an email address and a user wanted to change their email address, you would have to create a new row with the new email address and copy all of the existing columns from the old row to the new row.

UUID Types for Column Names

The UUID comparator type (universally unique id) is used to avoid collisions in column names. For example, if you wanted to identify a column (such as a blog entry or a tweet) by its timestamp, multiple clients writing to the same row key simultaneously could cause a timestamp collision, potentially overwriting data that was not intended to be overwritten. Using the UUIDType to represent a type-1 (time-based) UUID can avoid such collisions.

Managing and Accessing Data in Cassandra

This section provides information about accessing and managing data in Cassandra via a client application. Cassandra offers a number of client utilities and application programming interfaces (APIs) that can be used for developing applications that utilize Cassandra for data storage and retrieval.

About Writes in Cassandra

Cassandra is optimized for very fast and highly available data writing. Relational databases typically structure tables in order to keep data duplication at a minimum. The various pieces of information needed to satisfy a query are stored in various related tables that adhere to a pre-defined structure. Because of the way data is structured in a relational database, writing data is expensive, as the database server has to do additional work to ensure data integrity across the various related tables. As a result, relational databases usually are not performant on writes.

Cassandra is optimized for write throughput. Cassandra writes are first written to a commit log (for durability), and then to an in-memory table structure called a *memtable*. A write is successful once it is written to the commit log and memory, so there is very minimal disk I/O at the time of write. Writes are batched in memory and periodically written to disk to a persistent table structure called an *SSTable* (sorted string table). Memtables and SSTables are maintained per column family. Memtables are organized in sorted order by row key and flushed to SSTables sequentially (no random seeking as in relational databases).

SSTables are immutable (they are not written to again after they have been flushed). This means that a row is typically stored across multiple SSTable files. At read time, a row must be combined from all SSTables on disk (as well as unflushed memtables) to produce the requested data. To optimize this piecing-together process, Cassandra uses an in-memory structure called a *bloom filter*. Each SSTable has a bloom filter associated with it. The bloom filter is used to check if a requested row key exists in the SSTable before doing any disk seeks.

For a detailed explanation of how client read and write requests are handled in Cassandra, also see *About Client Requests in Cassandra*.

About Compaction

In the background, Cassandra periodically merges SSTables together into larger SSTables using a process called *compaction*. Compaction merges row fragments together, removes expired tombstones (deleted columns), and rebuilds primary and secondary indexes. Since the SSTables are sorted by row key, this merge is efficient (no random disk I/O). Once a newly merged SSTable is complete, the input SSTables are marked as obsolete and eventually deleted by the JVM garbage collection (GC) process. However, during compaction, there is a temporary spike in disk space usage and disk I/O.

Compaction impacts read performance in two ways. While a compaction is in progress, it temporarily increases disk I/O and disk utilization which can impact read performance for reads that are not fulfilled by the cache. However, after a compaction has been completed, off-cache read performance improves since there are fewer SSTable files on disk that need to be checked in order to complete a read request.

As of Cassandra 1.0, there are two different compaction strategies that you can configure on a column family - size-tiered compaction or leveled compaction. See *Tuning Compaction* for a description of these compaction strategies.

About Transactions and Concurrency Control

Unlike relational databases, Cassandra does not offer fully ACID-compliant transactions. There is no locking or transactional dependencies when concurrently updating multiple rows or column families.

ACID is an acronym used to describe transactional behavior in a relational database systems, which stands for:

- **Atomic**. Everything in a transaction succeeds or the entire transaction is rolled back.
- **Consistent**. A transaction cannot leave the database in an inconsistent state.
- **Isolated**. Transactions cannot interfere with each other.
- **Durable**. Completed transactions persist in the event of crashes or server failure.

Cassandra trades transactional isolation and atomicity for high availability and fast write performance. In Cassandra, a write is atomic at the row-level, meaning inserting or updating columns for a given row key will be treated as one write operation. Cassandra does not support transactions in the sense of bundling multiple row updates into one all-or-nothing operation. Nor does it roll back when a write succeeds on one replica, but fails on other replicas. It is possible in Cassandra to have a write operation report a failure to the client, but still actually persist the write to a replica.

For example, if using a write consistency level of QUORUM with a replication factor of 3, Cassandra will send the write to 2 replicas. If the write fails on one of the replicas but succeeds on the other, Cassandra will report a write failure to the client. However, the write is not automatically rolled back on the other replica.

Cassandra uses timestamps to determine the most recent update to a column. The timestamp is provided by the client application. The latest timestamp always wins when requesting data, so if multiple client sessions update the same columns in a row concurrently, the most recent update is the one that will eventually persist.

Writes in Cassandra are durable. All writes to a replica node are recorded both in memory and in a commit log before they are acknowledged as a success. If a crash or server failure occurs before the memory tables are flushed to disk, the commit log is replayed on restart to recover any lost writes.

About Inserts and Updates

Any number of columns may be inserted at the same time. When inserting or updating columns in a column family, the client application specifies the row key to identify which column records to update. The row key is similar to a primary key in that it must be unique for each row within a column family. However, unlike a primary key, inserting a duplicate row key will not result in a primary key constraint violation - it will be treated as an `UPSERT` (update the specified columns in that row if they exist or insert them if they do not).

Columns are only overwritten if the timestamp in the new version of the column is more recent than the existing column, so precise timestamps are necessary if updates (overwrites) are frequent. The timestamp is provided by the client, so the clocks of all client machines should be synchronized using NTP (network time protocol).

About Deletes

When deleting a row or a column in Cassandra, there are a few things to be aware of that may differ from what one would expect in a relational database.

1. **Deleted data is not immediately removed from disk.** Data that is inserted into Cassandra is persisted to SSTables on disk. Once an SSTable is written, it is immutable (the file is not updated by further DML operations). This means that a deleted column is not removed immediately. Instead a marker called a *tombstone* is written to indicate the new column status. Columns marked with a tombstone exist for a configured time period (defined by the *gc_grace_seconds* value set on the column family), and then are permanently deleted by the compaction process after that time has expired.
2. **A deleted column can reappear if routine node repair is not run.** Marking a deleted column with a tombstone ensures that a replica that was down at the time of delete will eventually receive the delete when it comes back up again. However, if a node is down longer than the configured time period for keeping tombstones (defined by the *gc_grace_seconds* value set on the column family), then the node can possibly miss the delete altogether, and replicate deleted data once it comes back up again. To prevent deleted data from reappearing, administrators must run regular node repair on every node in the cluster (by default, every 10 days).

3. **The row key for a deleted row may still appear in range query results.** When you delete a row in Cassandra, it marks all columns for that row key with a tombstone. Until those tombstones are cleared by compaction, you have an empty row key (a row that contains no columns). These deleted keys can show up in results of `get_range_slices()` calls. If your client application performs range queries on rows, you may want to have if filter out row keys that return empty column lists.

About Hinted Handoff Writes

Hinted handoff is an optional feature of Cassandra that reduces the time to restore a failed node to consistency once the failed node returns to the cluster. It can also be used for absolute write availability for applications that cannot tolerate a failed write, but can tolerate inconsistent reads.

When a write is made, Cassandra attempts to write to all replicas for the affected row key. If a replica is known to be down at the time the write occurs, a corresponding live replica will store a hint. The hint consists of location information (the replica node and row key that require a replay), as well as the actual data being written. There is minimal overhead to storing hints on replica nodes that already own the written row, since the data being written is already accounted for by the usual write process. The hint data itself is relatively small in comparison to most data rows.

If all replicas for the affected row key are down, it is still possible for a write to succeed if using a *write consistency* level of ANY. Under this scenario, the hint and written data are stored on the coordinator node, but will not be available to reads until the hint gets written to the actual replicas that own the row. The ANY consistency level provides absolute write availability at the cost of consistency, as there is no guarantee as to when written data will be available to reads (depending how long the replicas are down). Using the ANY consistency level can also potentially increase load on the cluster, as coordinator nodes must temporarily store extra rows whenever a replica is not available to accept a write.

Note

By default, hints are only saved for one hour before they are dropped. If all replicas are down at the time of write, and they all remain down for longer than the configured time of *max_hint_window_in_ms*, you could potentially lose a write made at consistency level ANY.

Hinted handoff does not count towards any other consistency level besides ANY. For example, if using a consistency level of ONE and all replicas for the written row are down, the write will fail regardless of whether a hint is written or not.

When a replica that is storing hints detects via gossip that the failed node is alive again, it will begin streaming the missed writes to catch up the out-of-date replica.

Note

Hinted handoff does not completely replace the need for regular node repair operations.

About Reads in Cassandra

When a read request for a row comes in to a node, the row must be combined from all SSTables on that node that contain columns from the row in question, as well as from any unflushed memtables, to produce the requested data. To optimize this piecing-together process, Cassandra uses an in-memory structure called a bloom filter: each SSTable has a bloom filter associated with it that is used to check if any data for the requested row exists in the SSTable before doing any disk I/O. As a result, Cassandra is very performant on reads when compared to other storage systems, even for read-heavy workloads.

As with any database, reads are fastest when the most in-demand data (or *hot* working set) fits into memory. Although all modern storage systems rely on some form of caching to allow for fast access to hot data, not all of them degrade gracefully when the cache capacity is exceeded and disk I/O is required. Cassandra's read performance benefits from built-in caching, but it also does not dip dramatically when random disk seeks are required. When I/O activity starts to increase in Cassandra due to increased read load, it is easy to remedy by adding more nodes to the cluster.

For rows that are accessed frequently, Cassandra has a built-in key cache (and an optional row cache). See *Tuning the Cache* for more information about optimizing read performance using the built-in caching features.

For a detailed explanation of how client read and write requests are handled in Cassandra, also see *About Client Requests in Cassandra*.

About Data Consistency in Cassandra

In Cassandra, consistency refers to how up-to-date and synchronized a row of data is on all of its replicas. Cassandra extends the concept of eventual consistency by offering *tunable consistency*. For any given read or write operation, the client application decides how consistent the requested data should be.

In addition to tunable consistency, Cassandra has a number of *built-in repair mechanisms* to ensure that data remains consistent across replicas.

Tunable Consistency for Client Requests

Consistency levels in Cassandra can be set on any read or write query. This allows application developers to tune consistency on a per-query basis depending on their requirements for response time versus data accuracy. Cassandra offers a number of consistency levels for both reads and writes.

About Write Consistency

When you do a write in Cassandra, the consistency level specifies on how many replicas the write must succeed before returning an acknowledgement to the client application.

The following consistency levels are available, with ANY being the lowest consistency (but highest availability), and ALL being the highest consistency (but lowest availability). QUORUM is a good middle-ground ensuring strong consistency, yet still tolerating some level of failure.

A quorum is calculated as (rounded down to a whole number):

```
(replication_factor / 2) + 1
```

For example, with a replication factor of 3, a quorum is 2 (can tolerate 1 replica down). With a replication factor of 6, a quorum is 4 (can tolerate 2 replicas down).

Level	Description
ANY	A write must be written to at least one node. If all replica nodes for the given row key are down, the write can still succeed once a *hinted handoff* has been written. Note that if all replica nodes are down at write time, an ANY write will not be readable until the replica nodes for that row key have recovered.
ONE	A write must be written to the commit log and memory table of at least one replica node.
QUORUM	A write must be written to the commit log and memory table on a quorum of replica nodes.
LOCAL_QUORUM	A write must be written to the commit log and memory table on a quorum of replica nodes in the same data center as the coordinator node. Avoids latency of inter-data center communication.
EACH_QUORUM	A write must be written to the commit log and memory table on a quorum of replica nodes in *all* data centers.
ALL	A write must be written to the commit log and memory table on all replica nodes in the cluster for that row key.

About Read Consistency

When you do a read in Cassandra, the consistency level specifies how many replicas must respond before a result is returned to the client application.

Cassandra checks the specified number of replicas for the most recent data to satisfy the read request (based on the timestamp).

The following consistency levels are available, with ONE being the lowest consistency (but highest availability), and ALL being the highest consistency (but lowest availability). QUORUM is a good middle-ground ensuring strong consistency, yet still tolerating some level of failure.

A quorum is calculated as (rounded down to a whole number):

```
(replication_factor / 2) + 1
```

For example, with a replication factor of 3, a quorum is 2 (can tolerate 1 replica down). With a replication factor of 6, a quorum is 4 (can tolerate 2 replicas down).

Level	Description
ONE	Returns a response from the closest replica (as determined by the snitch). By default, a read repair runs in the background to make the other replicas consistent.
QUORUM	Returns the record with the most recent timestamp once a quorum of replicas has responded.
LOCAL_QUORUM	Returns the record with the most recent timestamp once a quorum of replicas in the current data center as the coordinator node has reported. Avoids latency of inter-data center communication.
EACH_QUORUM	Returns the record with the most recent timestamp once a quorum of replicas in each data center of the cluster has responded.
ALL	Returns the record with the most recent timestamp once all replicas have responded. The read operation will fail if a replica does not respond.

Note

LOCAL_QUORUM and EACH_QUORUM are designed for use in multi-data center clusters using a rack-aware replica placement strategy (such as `NetworkTopologyStrategy`) and a properly configured snitch.

Choosing Client Consistency Levels

Choosing a consistency level for reads and writes involves determining your requirements for consistent results (always reading the most recently written data) versus read or write latency (the time it takes for the requested data to be returned or for the write to succeed).

If latency is a top priority, consider a consistency level of ONE (only one replica node must successfully respond to the read or write request). There is a higher probability of stale data being read with this consistency level (as the replicas contacted for reads may not always have the most recent write). For some applications, this may be an acceptable trade-off. If it is an absolute requirement that a write never fail, you may also consider a write consistency level of ANY. This consistency level has the highest probability of a read not returning the latest written values (see *hinted handoff*).

If consistency is top priority, you can ensure that a read will always reflect the most recent write by using the following formula:

```
(nodes_written + nodes_read) > replication_factor
```

For example, if your application is using the QUORUM consistency level for both write and read operations and you are using a replication factor of 3, then this ensures that 2 nodes are always written and 2 nodes are always read. The combination of nodes written and read (4) being greater than the replication factor (3) ensures strong read consistency.

Consistency Levels for Multi-Data Center Clusters

A client read or write request to a Cassandra cluster always specifies the *consistency level* it requires. Ideally, you want a client request to be served by replicas in the same data center in order to avoid latency. Contacting multiple data centers for a read or write request can slow down the response. The consistency level `LOCAL_QUORUM` is specifically designed for doing quorum reads and writes in multi data center clusters.

A consistency level of `ONE` is also fine for applications with less stringent consistency requirements. A majority of Cassandra users do writes at consistency level `ONE`. With this consistency, the request will always be served by the replica node closest to the coordinator node that received the request (unless the *dynamic snitch* determines that the node is performing poorly and routes it elsewhere).

Keep in mind that even at consistency level `ONE` or `LOCAL_QUORUM`, the write is still sent to all replicas for the written key, even replicas in other data centers. The consistency level just determines how many replicas are required to respond that they received the write.

Specifying Client Consistency Levels

Consistency level is specified by the client application when a read or write request is made. The default consistency level may differ depending on the client you are using.

For example, in CQL the default consistency level for reads and writes is ONE. If you wanted to use QUORUM instead, you could specify that consistency level in the client request as follows:

```
SELECT * FROM users WHERE state='TX' USING CONSISTENCY QUORUM;
```

About Cassandra's Built-in Consistency Repair Features

Cassandra has a number of built-in repair features to ensure that data remains consistent across replicas. These features are:

- **Read Repair** - For reads, there are two types of read requests that a coordinator can send to a replica; a direct read request and a background *read repair* request. The number of replicas contacted by a direct read request is determined by the *consistency level* specified by the client. Background read repair requests are sent to any additional replicas that did not receive a direct request. To ensure that frequently-read data remains consistent, the coordinator compares the data from all the remaining replicas that own the row in the background, and if they are inconsistent, issues writes to the out-of-date replicas to update the row to reflect the most recently written values. Read repair can be configured per column family (using *read_repair_chance*), and is enabled by default.
- **Anti-Entropy Node Repair** - For data that is not read frequently, or to update data on a node that has been down for a while, the *nodetool repair* process (also referred to as anti-entropy repair) ensures that all data on a replica is made consistent. Node repair should be run routinely as part of regular cluster maintenance operations.
- **Hinted Handoff** - Writes are always sent to all replicas for the specified row regardless of the consistency level specified by the client. If a node happens to be down at the time of write, its corresponding replicas will save hints about the missed writes, and then handoff the affected rows once the node comes back online. Hinted handoff ensures data consistency due to short, transient node outages. The hinted handoff feature is configurable at the node-level in the *cassandra.yaml file* See *About Hinted Handoff Writes* for more information on how hinted handoff works.

Cassandra Client APIs

When Cassandra was first released, it originally provided a Thrift RPC-based API as the foundation for client developers to build upon. This proved to be suboptimal: Thrift is too low-level to use without a more idiomatic client wrapping it, and supporting new features (such as secondary indexes in 0.7 and counters in 0.8) became hard to maintain across these clients for many languages. Also, by not having client development hosted within the Apache Cassandra project itself, incompatible clients proliferated in the open source community, all with different levels of stability and features. It became hard for application developers to choose the best API to fit their needs.

About Cassandra CLI

Cassandra 0.7 introduced a stable version of its command-line client interface, `cassandra-cli`, that can be used for common data definition (DDL), data manipulation (DML), and data exploration. Although not intended for application development, it is a good way to get started defining your data model and becoming familiar with Cassandra.

About CQL

Cassandra 0.8 was the first release to include the Cassandra Query Language (CQL). As with SQL, clients built on CQL only need to know how to interpret query `resultset` objects. CQL is the future of Cassandra client API development. CQL drivers are hosted within the Apache Cassandra project.

Note

CQL version 2.0, which has improved support for several commands, is compatible with Cassandra version 1.0 but not version 0.8.x.

CQL syntax in based on SQL (Structured Query Language), the standard for relational database manipulation. Although CQL has many similarities to SQL, it does not change the underlying Cassandra data model. There is no support for JOINs, for example.

The Python driver includes a command-line interface, `cql.sh`. See *Getting Started with CQL*.

Other High-Level Clients

The Thrift API will continue to be supported for backwards compatibility. Using a high-level client is highly recommended over using raw Thrift calls.

A list of other available clients may be found on the Client Options page.

The Java, Python, and PHP clients are well supported.

Java: Hector Client API

Hector provides Java developers with features lacking in Thrift, including connection pooling, JMX integration, failover and extensive logging. Hector is the first client to implement CQL.

For more information, see the Hector web site.

Python: Pycassa Client API

Pycassa is a Python client API with features such as connection pooling, SuperColumn support, and a method to map existing classes to Cassandra column families.

For more information, see the Pycassa documentation.

PHP: Phpcassa Client API

Phpcassa is a PHP client API with features such as connection pooling, a method for counting rows, and support for secondary indexes.

For more information, see the Phpcassa documentation.

Getting Started Using the Cassandra CLI

The Cassandra CLI client utility can be used to do basic data definition (DDL) and data manipulation (DML) within a Cassandra cluster. It is located in `/usr/bin/cassandra-cli` in packaged installations or `$CASSANDRA_HOME/bin/cassandra-cli` in binary installations.

To start the CLI and connect to a particular Cassandra instance, launch the script together with `-host` and `-port` options. It will connect to the cluster name specified in the *cassandra.yaml* file (which is *Test Cluster*` by default). For example, if you have a single-node cluster on `localhost`:

```
$ cassandra-cli -host localhost -port 9160
```

Or to connect to a node in a multi-node cluster, give the IP address of the node:

```
$ cassandra-cli -host 110.123.4.5 -port 9160
```

To see help on the various commands available:

```
[default@unknown] help;
```

For detailed help on a specific command, use `help <command>;`. For example:

```
[default@unknown] help SET;
```

Note

A command is not sent to the server unless it is terminated by a semicolon (`;`). Hitting the return key without a semicolon at the end of the line echos an ellipsis (`. . .`), which indicates that the CLI expects more input.

Creating a Keyspace

You can use the Cassandra CLI commands described in this section to create a keyspace. In this example, we create a keyspace called `demo`, with a replication factor of 1 and using the `SimpleStrategy` replica placement strategy.

Note the single quotes around the string value of `placement_strategy`:

```
[default@unknown] CREATE KEYSPACE demo
with placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy'
and strategy_options = [{replication_factor:1}];
```

You can verify the creation of a keyspace with the `SHOW KEYSPACES` command. The new keyspace is listed along with the `system` keyspace and any other existing keyspaces.

Creating a Column Family

First, connect to the keyspace where you want to define the column family with the `USE` command.

```
[default@unknown] USE demo;
```

In this example, we create a `users` column family in the `demo` keyspace. In this column family we are defining a few columns; `full_name`, `email`, `state`, `gender`, and `birth_year`. This is considered a *static* column family - we are defining the column names up front and most rows are expected to have more-or-less the same columns.

Notice the settings of `comparator`, `key_validation_class` and `validation_class`. These are setting the default encoding used for column names, row key values and column values. In the case of column names, the comparator also determines the sort order.

```
[default@unknown] USE demo;

[default@demo] CREATE COLUMN FAMILY users
WITH comparator = UTF8Type
AND key_validation_class=UTF8Type
AND column_metadata = [
{column_name: full_name, validation_class: UTF8Type}
{column_name: email, validation_class: UTF8Type}
{column_name: state, validation_class: UTF8Type}
{column_name: gender, validation_class: UTF8Type}
{column_name: birth_year, validation_class: LongType}
];
```

Next, create a *dynamic* column family called `blog_entry`. Notice that here we do not specify column definitions as the column names are expected to be supplied later by the client application.

```
[default@demo] CREATE COLUMN FAMILY blog_entry
WITH comparator = TimeUUIDType
AND key_validation_class=UTF8Type
AND default_validation_class = UTF8Type;
```

Creating a Counter Column Family

A counter column family contains counter columns. A counter column is a specific kind of column whose user-visible value is a 64-bit signed integer that can be incremented (or decremented) by a client application. The counter column tracks the most recent value (or count) of all updates made to it. A counter column cannot be mixed in with regular columns of a column family, you must create a column family specifically to hold counters.

To create a column family that holds counter columns, set the `default_validation_class` of the column family to `CounterColumnType`. For example:

```
[default@demo] CREATE COLUMN FAMILY page_view_counts
WITH default_validation_class=CounterColumnType
AND key_validation_class=UTF8Type AND comparator=UTF8Type;
```

To insert a row and counter column into the column family (with the initial counter value set to 0):

```
[default@demo] INCR page_view_counts['www.datastax.com'][home] BY 0;
```

To increment the counter:

```
[default@demo] INCR page_view_counts['www.datastax.com'][home] BY 1;
```

Inserting Rows and Columns

The following examples illustrate using the `SET` command to insert columns for a particular row key into the `users` column family. In this example, the row key is `bobbyjo` and we are setting each of the columns for this user. Notice that you can only set one column at a time in a `SET` command.

```
[default@demo] SET users['bobbyjo']['full_name']='Robert Jones';

[default@demo] SET users['bobbyjo']['email']='bobjones@gmail.com';

[default@demo] SET users['bobbyjo']['state']='TX';

[default@demo] SET users['bobbyjo']['gender']='M';

[default@demo] SET users['bobbyjo']['birth_year']='1975';
```

In this example, the row key is `yomama` and we are just setting some of the columns for this user.

```
[default@demo] SET users['yomama']['full_name']='Cathy Smith';

[default@demo] SET users['yomama']['state']='CA';

[default@demo] SET users['yomama']['gender']='F';

[default@demo] SET users['yomama']['birth_year']='1969';
```

In this example, we are creating an entry in the `blog_entry` column family for row key `yomama`:

```
[default@demo] SET blog_entry['yomama'][timeuuid()] = 'I love my new shoes!';
```

Note

The Cassandra CLI uses a default consistency level of ONE for all write and read operations. Specifying different consistency levels is not supported within Cassandra CLI.

Reading Rows and Columns

Use the `GET` command within Cassandra CLI to retrieve a particular row from a column family. Use the `LIST` command to return a batch of rows and their associated columns (default limit of rows returned is 100).

For example, to return the first 100 rows (and all associated columns) from the `users` column family:

```
[default@demo] LIST users;
```

Cassandra stores all data internally as hex byte arrays by default. If you do not specify a default row key validation class, column comparator and column validation class when you define the column family, Cassandra CLI will expect input data for row keys, column names, and column values to be in hex format (and data will be returned in hex format).

To pass and return data in human-readable format, you can pass a value through an encoding function. Available encodings are:

- ascii
- bytes
- integer (a generic variable-length integer type)
- lexicalUUID
- long
- utf8

For example to return a particular row key and column in UTF8 format:

```
[default@demo] GET users[utf8('bobby')][utf8('full_name')];
```

You can also use the `ASSUME` command to specify the encoding in which column family data should be returned for the entire client session. For example, to return row keys, column names, and column values in ASCII-encoded format:

```
[default@demo] ASSUME users KEYS AS ascii;
[default@demo] ASSUME users COMPARATOR AS ascii;
[default@demo] ASSUME users VALIDATOR AS ascii;
```

Setting an Expiring Column

When you set a column in Cassandra, you can optionally set an expiration time, or *time-to-live* (TTL) attribute for it.

For example, suppose we are tracking coupon codes for our users that expire after 10 days. We can define a `coupon_code` column and set an expiration date on that column. For example:

```
[default@demo] SET users['bobbyjo']
[utf8('coupon_code')] = utf8('SAVE20') WITH ttl=864000;
```

After ten days, or 864,000 seconds have elapsed since the setting of this column, its value will be marked as deleted and no longer be returned by read operations. Note, however, that the value is not actually deleted from disk until normal Cassandra compaction processes are completed.

Indexing a Column

The CLI can be used to create secondary indexes (indexes on column values). You can add a secondary index when you create a column family or add it later using the `UPDATE COLUMN FAMILY` command.

For example, to add a secondary index to the `birth_year` column of the `users` column family:

```
[default@demo] UPDATE COLUMN FAMILY users
WITH comparator = UTF8Type
AND column_metadata = [{column_name: birth_year, validation_class: LongType, index_type: KEYS}];
```

Because of the secondary index created for the column `birth_year`, its values can be queried directly for users born in a given year as follows:

```
[default@demo] GET users WHERE birth_date = 1969;
```

Deleting Rows and Columns

The Cassandra CLI provides the `DEL` command to delete a row or column (or subcolumn).

For example, to delete the `coupon_code` column for the `yomama` row key in the `users` column family:

```
[default@demo] DEL users ['yomama']['coupon_code'];

[default@demo] GET users ['yomama'];
```

Or to delete an entire row:

```
[default@demo] DEL users ['yomama'];
```

Dropping Column Families and Keyspaces

With Cassandra CLI commands you can drop column families and keyspaces in much the same way that tables and databases are dropped in a relational database. This example shows the commands to drop our example `users` column family and then drop the `demo` keyspace altogether:

```
[default@demo] DROP COLUMN FAMILY users;

[default@demo] DROP KEYSPACE demo;
```

Getting Started with CQL

Developers can access CQL commands in a variety of ways. Drivers are available for Python, Twisted Python, and JDBC-based client programs.

For the purposes of administrators, the most direct way to run simple CQL commands is via the Python-based `cqlsh` command-line client.

Starting the CQL Command-Line Program (cqlsh)

As of Apache Cassandra version 1.0.5 and DataStax Community version 1.0.1, the `cqlsh` client is installed with Cassandra in `$CASSANDRA_HOME/bin/cqlsh` for tarball installations, or `/usr/bin/cqlsh` for packaged installations.

When you start `cqlsh`, you must provide the IP of a Cassandra node to connect to (default is `localhost`) and the RPC connection port (default is 9160). For example:

```
$ cqlsh 103.263.89.126 9160
cqlsh>
```

To exit `cqlsh` type `exit` at the command prompt.

```
cqlsh> exit
```

Running CQL Commands with cqlsh

Commands in `cqlsh` combine SQL-like syntax that maps to Cassandra concepts and operations. If you are just getting started with CQL, make sure to refer to the *CQL Reference*.

As of CQL version 2.0, `cqlsh` has the following limitations in support for Cassandra operations and data objects:

- Super Columns are not supported; column_type and subcomparator arguments are not valid
- Composite columns are not supported
- Only a subset of all the available column family storage properties can be set using CQL.

The rest of this section provides some guidance with simple CQL commands using `cqlsh`. This is a similar (but not identical) set of commands as the set described in *Using the Cassandra Client*.

Creating a Keyspace

You can use the `cqlsh` commands described in this section to create a keyspace. In creating an example keyspace for Twissandra, we will assume a desired replication factor of 3 and implementation of the NetworkTopologyStrategy replica placement strategy. For more information on these keyspace options, see *About Replication in Cassandra*.

Note the single quotes around the string value of `strategy_class`:

```
cqlsh> CREATE KEYSPACE twissandra WITH
       strategy_class = 'NetworkTopologyStrategy'
       AND strategy_options:DC1 = 3;
```

Creating a Column Family

For this example, we use `cqlsh` to create a *users* column family in the newly created keyspace. Note the `USE` command to connect to the twissandra keyspace.

```
cqlsh> USE twissandra;

cqlsh> CREATE COLUMNFAMILY users (
 ...  KEY varchar PRIMARY KEY,
 ...  password varchar,
 ...  gender varchar,
 ...  session_token varchar,
 ...  state varchar,
 ...  birth_year bigint);
```

Inserting and Retrieving Columns

Though in production scenarios it is more practical to insert columns and column values programmatically, it is possible to use `cqlsh` for these operations. The example in this section illustrates using the `INSERT` and `SELECT` commands to insert and retrieve some columns in the *users* column family.

The following commands create and then get a user record for "jsmith." The record includes a value for the password column we created when we created the column family, as well as an expiration time for the password column. Note that the user name "jsmith" is the row key, or in CQL terms, the primary key.

```
cqlsh> INSERT INTO users (KEY, password) VALUES ('jsmith', 'ch@ngem3a') USING TTL 86400;
cqlsh> SELECT * FROM users WHERE KEY='jsmith';
u'jsmith' | u'password',u'ch@ngem3a' | u'ttl', 86400
```

Adding Columns with ALTER COLUMNFAMILY

The `ALTER COLUMNFAMILY` command lets you add new columns to a column family. For example, to add a `coupon_code` column with the `varchar` validation type to the `users` column family:

```
cqlsh> ALTER TABLE users ADD coupon_code varchar;
```

This creates the column metadata and adds the column to the column family schema, but does not update any existing rows.

Altering Column Metadata

With ALTER COLUMNFAMILY, you can change the type of a column any time after it is defined or added to a column family. For example, if we decided the `coupon_code` column should store coupon codes in the form of integers, we could change the validation type as follows:

```
cqlsh> ALTER TABLE users ALTER coupon_code TYPE int;
```

Note that existing coupon codes will not be validated against the new type, only newly inserted values.

Specifying Column Expiration with TTL

Both the `INSERT` and `UPDATE` commands support setting a column expiration time (TTL). In the `INSERT` example above for the key `jsmith` we set the password column to expire at 86400 seconds, or one day. If we wanted to extend the expiration period to five days, we could use the `UPDATE` command a shown:

```
cqlsh> UPDATE users USING TTL 432000 SET 'password' = 'ch@ngem3a' WHERE KEY = 'jsmith';
```

Dropping Column Metadata

If your aim is to remove a column's metadata entirely, including the column name and validation type, you can use `ALTER TABLE <columnFamily> DROP <column>`. The following command removes the name and validator without affecting or deleting any existing data:

```
cqlsh> ALTER TABLE users DROP coupon_code;
```

After you run this command, clients can still add new columns named `coupon_code` to the `users` column family -- but they will not be validated until you explicitly add a type again.

Indexing a Column

`cqlsh` can be used to create secondary indexes, or indexes on column values. In this example, we will create an index on the `state` and `birth_year` columns in the users column family.

```
cqlsh> CREATE INDEX state_key ON users (state);
cqlsh> CREATE INDEX birth_year_key ON users (birth_year);
```

Because of the secondary index created for the two columns, their values can be queried directly as follows:

```
cqlsh> SELECT * FROM users
 ... WHERE gender='f' AND
 ...  state='TX' AND
...  birth_year='1968';
u'user1' | u'birth_year',1968 | u'gender',u'f' | u'password',u'ch@ngem3' | u'state',u'TX'
```

Deleting Columns and Rows

`cqlsh` provides the `DELETE` command to delete a column or row. In this example we will delete user jsmith's session token column, and then delete jsmith's row entirely.

```
cqlsh> DELETE session_token FROM users where KEY = 'jsmith';
cqlsh> DELETE FROM users where KEY = 'jsmith';
```

Note, however, that the phenomena called "range ghosts" in Cassandra may mean that keys for deleted rows are still retrieved by `SELECT` statements and other "get" operations. Deleted values, including range ghosts, are removed completely by the first compaction following deletion.

Dropping Column Families and Keyspaces

With `cqlsh` commands you can drop column families and keyspaces in much the same way that tables and databases are dropped in relational models. This example shows the commands to drop our example *users* column family and then drop the *twissandra* keyspace altogether:

```
cqlsh> DROP COLUMNFAMILY users;
cqlsh> DROP KEYSPACE twissandra;
```

Configuration

Like any modern server-based software, Cassandra has a number of configuration options to tune the system towards specific workloads and environments. Substantial efforts have been made to provide meaningful default configuration values, but given the inherently complex nature of distributed systems coupled with the wide variety of possible workloads, most production deployments will require some modifications of the default configuration.

Node and Cluster Configuration (cassandra.yaml)

The `cassandra.yaml` file is the main configuration file for Cassandra. This file is located in `/etc/cassandra/conf/cassandra.yaml` in packaged installations or `$CASSANDRA_HOME/conf/cassandra.yaml` in binary installations. After changing properties in this file, you must restart the node for the changes to take effect.

Option	Default Value
authenticator	org.apache.cassandra.auth.AllowAllAuthenticator
authority	org.apache.cassandra.auth.AllowAllAuthority
auto_bootstrap	false
broadcast_address	same as listen_address
cluster_name	Test Cluster
column_index_size_in_kb	64
commitlog_directory	/var/lib/cassandra/commitlog
commitlog_sync	periodic
commitlog_sync_period_in_ms	10000 (ten seconds)
commitlog_total_space_in_mb	4096
compaction_preheat_key_cache	true
compaction_throughput_mb_per_sec	16
concurrent_compactors	One per CPU core
concurrent_reads	32
concurrent_writes	32
data_file_directories	/var/lib/cassandra/data
dynamic_snitch	true
dynamic_snitch_badness_threshold	0.0
dynamic_snitch_reset_interval_in_ms	600000

dynamic_snitch_update_interval_in_ms	100
endpoint_snitch	org.apache.cassandra.locator.SimpleSnitch
flush_largest_memtables_at	0.75
hinted_handoff_enabled	true
hinted_handoff_throttle_delay_in_ms	50
in_memory_compaction_limit_in_mb	64
incremental_backups	false
index_interval	128
initial_token	n/a
internode_encryption	none
keystore	conf/.keystore
keystore_password	cassandra
listen_address	localhost
max_hint_window_in_ms	3600000 (one hour)
memtable_flush_queue_size	4
memtable_flush_writers	One per data directory
memtable_total_space_in_mb	1/3 of the heap
multithreaded_compaction	false
partitioner	org.apache.cassandra.dht.RandomPartitioner
phi_convict_threshold	8
reduce_cache_capacity_to	0.6
reduce_cache_sizes_at	0.85
request_scheduler	org.apache.cassandra.scheduler.NoScheduler
request_scheduler_id	keyspace
rpc_address	localhost
rpc_keepalive	true
rpc_max_threads	Unlimited
rpc_min_threads	16
rpc_port	9160
rpc_recv_buff_size_in_bytes	n/a
rpc_send_buff_size_in_bytes	n/a
rpc_server_type	sync
rpc_timeout_in_ms	10000
saved_caches_directory	/var/lib/cassandra/saved_caches
seeds	127.0.0.1
seed_provider	org.apache.cassandra.locator.SimpleSeedProvider
sliced_buffer_size_in_kb	64

snapshot_before_compaction	false
storage_port	700
stream_throughput_outbound_megabits_per_sec	400
thrift_framed_transport_size_in_mb	15
thrift_max_message_length_in_mb	16
truststore	conf/.truststore
truststore_password	cassandra

Node and Cluster Initialization Properties

The following properties are used to initialize a new cluster or when introducing a new node to an established cluster, and should be evaluated and changed as needed before starting a node for the first time. These properties control how a node is configured within a cluster in regards to inter-node communication, data partitioning, and replica placement.

auto_bootstrap

When set to true, populates a new node with a range of data when it joins an established cluster based on the setting of initial_token. If initial_token is not set, the newly added node will insert itself into the ring by splitting the token range of the most heavily loaded node. Leave set to false when initializing a brand new cluster.

broadcast_address

If your Cassandra cluster is deployed across multiple Amazon EC2 regions (and you are using the *EC2MultiRegionSnitch*), you should set broadcast_address to *public* IP address of the node (and listen_address to the *private* IP). If not declared, defaults to the same address as specified for listen_address.

cluster_name

The name of the cluster. All nodes participating in a cluster must have the same value.

commitlog_directory

The directory where the commit log will be stored. For optimal write performance, DataStax recommends the commit log be on a separate disk partition (ideally a separate physical device) from the data file directories.

data_file_directories

The directory location where column family data (SSTables) will be stored.

initial_token

The initial token assigns the node token position in the ring, and assigns a range of data to the node when it first starts up. The initial token can be left unset when introducing a new node to an established cluster using auto_bootstrap. Otherwise, the token value depends on the partitioner you are using. With the random partitioner, this value will be a number between 0 and 2**127. With the byte order preserving partitioner, this value will be a byte array of hex values based on your actual row key values. With the order preserving and collated order preserving partitioners, this value will be a UTF-8 string based on your actual row key values. See *Calculating Tokens* for more information.

listen_address

The IP address or hostname that other Cassandra nodes will use to connect to this node. If left blank, you must have hostname resolution correctly configured on all nodes in your cluster so that the hostname resolves to the correct IP address for this node (using `/etc/hostname`, `/etc/hosts` or DNS).

partitioner

Sets the partitioning method used when assigning a row key to a particular node (also see initial_token). Allowed values are:

- org.apache.cassandra.dht.RandomPartitioner (default)
- org.apache.cassandra.dht.ByteOrderedPartitioner
- org.apache.cassandra.dht.OrderPreservingPartitioner (deprecated)
- org.apache.cassandra.dht.CollatingOrderPreservingPartitioner (deprecated)

rpc_address

The listen address for remote procedure calls (client connections). To listen on all configured interfaces, set to 0.0.0.0. If left blank, you must have hostname resolution correctly configured on all nodes in your cluster so that the hostname resolves to the correct IP address for this node (using /etc/hostname, /etc/hosts or DNS). Default Value: localhost Allowed Values: An IP address, hostname, or leave unset to resolve the address using the hostname configuration of the node.

rpc_port

The port for remote procedure calls (client connections) and the Thrift service. Default is 9160.

saved_caches_directory

The directory location where column family key and row caches will be stored.

seed_provider

The seed provider is a pluggable interface for providing a list of seed nodes. The default seed provider requires a comma-delimited list of seeds.

seeds

When a node joins a cluster, it contacts the seed node(s) to determine the ring topology and obtain gossip information about the other nodes in the cluster. Every node in the cluster should have the same list of seeds, specified as a comma-delimited list of IP addresses. In multi data center clusters, the seed list should include at least one node from *each* data center (replication group).

storage_port

The port for inter-node communication. Default port is 7000.

endpoint_snitch

Sets the snitch to use for locating nodes and routing requests. In deployments with rack-aware replication placement strategies, use either RackInferringSnitch, PropertyFileSnitch, or EC2Snitch (if on Amazon EC2). Has a dependency on the replica *placement_strategy*, which is defined on a keyspace. The PropertyFileSnitch also requires a `cassandra-topology.properties` configuration file. Snitches included with Cassandra are:

- org.apache.cassandra.locator.SimpleSnitch
- org.apache.cassandra.locator.RackInferringSnitch
- org.apache.cassandra.locator.PropertyFileSnitch
- org.apache.cassandra.locator.EC2Snitch

Performance Tuning Properties

The following properties are used to tune performance and system resource utilization (memory, disk I/O, CPU, etc.) for reads and writes.

column_index_size_in_kb

Column indexes are added to a row after the data reaches this size. This usually happens if there are a large number of columns in a row or the column values themselves are large. If you consistently read only a few columns from each row, this should be kept small as it denotes how much of the row data must be deserialized to read the column.

commitlog_sync

The method that Cassandra will use to acknowledge writes. The default mode of periodic is used in conjunction with commitlog_sync_period_in_ms to control how often the commit log is synchronized to disk. Periodic syncs are acknowledged immediately. In batch mode, writes are not acknowledged until fsynced to disk. It will wait the configured number of milliseconds for other writes before performing a sync. Allowed Values are `periodic` (default) or `batch`.

commitlog_sync_period_in_ms

Determines how often (in milliseconds) to send the commit log to disk when commitlog_sync is set to `periodic` mode.

commitlog_total_space_in_mb

When the commitlog size on a node exceeds this threshold, Cassandra will flush memtables to disk for the oldest commitlog segments, thus allowing those log segments to be removed. This reduces the amount of data to replay on startup, and prevents infrequently-updated column families from keeping commit log segments around indefinitely. This replaces the per-column family storage setting `memtable_flush_after_mins`.

compaction_preheat_key_cache

When set to true, cached row keys are tracked during compaction, and re-cached to their new positions in the compacted SSTable. If you have extremely large key caches for your column families, set to false (see the *keys_cached* attribute set on a column family).

compaction_throughput_mb_per_sec

Throttles compaction to the given total throughput across the entire system. The faster you insert data, the faster you need to compact in order to keep the SSTable count down. The recommended Value is 16-32 times the rate of write throughput (in MBs/second). Setting to `0` disables compaction throttling.

concurrent_compactors

Sets the number of concurrent compaction processes allowed to run simultaneously on a node. Defaults to one compaction process per CPU core.

concurrent_reads

For workloads with more data than can fit in memory, the bottleneck will be reads that need to fetch data from disk. Setting to (16 * number_of_drives) allows operations to queue low enough in the stack so that the OS and drives can reorder them.

concurrent_writes

Writes in Cassandra are almost never I/O bound, so the ideal number of concurrent writes depends on the number of CPU cores in your system. The recommended value is (8 * number_of_cpu_cores).

flush_largest_memtables_at

When Java heap usage after a full concurrent mark sweep (CMS) garbage collection is higher than this percentage, the largest memtables will be flushed to disk in order to free memory. This parameter serves as more of an emergency measure for preventing sudden out-of-memory (OOM) errors rather than a strategic tuning mechanism. It is most effective under light to moderate load, or read-heavy workloads. The default value of .75 means flush memtables when Java heap usage is above 75 percent total heap size. 1.0 disables this feature.

in_memory_compaction_limit_in_mb

Size limit for rows being compacted in memory. Larger rows spill to disk and use a slower two-pass compaction process. When this occurs, a message is logged specifying the row key. The recommended value is 5 to 10 percent of the available Java heap size.

index_interval

Each SSTable has an index file containing row keys and the position at which that row starts in the data file. At startup, Cassandra reads a sample of that index into memory. By default 1 row key out of every 128 is sampled. To find a row, Cassandra performs a binary search on the sample, then does just one disk read of the index block corresponding to the closest sampled entry. The larger the sampling, the more effective the index is (at the cost of memory usage). A smaller value for this property results in a larger, more effective index. Generally, a value between 128 and 512 in combination with a large column family key cache offers the best trade off between memory usage and performance. You may want to increase the sample size if you have small rows, thus decreasing the index size and memory usage. For large rows, decreasing the sample size may improve read performance.

memtable_flush_queue_size

The number of full memtables to allow pending flush, that is, waiting for a writer thread. At a minimum, this should be set to the maximum number of secondary indexes created on a single column family.

memtable_flush_writers

Sets the number of memtable flush writer threads. These will be blocked by disk I/O, and each one will hold a memtable in memory while blocked. If you have a large Java heap size and many data directories (see data_file_directories), you can increase this value for better flush performance. By default this is set to the number of data directories defined (which is 1).

memtable_total_space_in_mb

Specifies total memory used for all column family memtables on a node. Defaults to a third of your JVM heap size. This replaces the old per-column family storage settings `memtable_operations_in_millions` and `memtable_throughput_in_mb`.

multithreaded_compaction

When set to true (it is false by default), each compaction operation will use one thread per SSTable being merged in addition to one thread per core. This is typically only useful on nodes with SSD hardware. With regular disks, the goal is to limit the disk I/O for compaction (see compaction_throughput_mb_per_sec).

reduce_cache_capacity_to

Sets the size percentage to which maximum cache capacity is reduced when Java heap usage reaches the threshold defined by reduce_cache_sizes_at. Together with flush_largest_memtables_at, these properties are an emergency measure for preventing sudden out-of-memory (OOM) errors.

reduce_cache_sizes_at

When Java heap usage after a full concurrent mark sweep (CMS) garbage collection is higher than this percentage, Cassandra will reduce the cache capacity to the fraction of the current size as specified by reduce_cache_capacity_to. The default is 85 percent (0.85). 1.0 disables this feature.

sliced_buffer_size_in_kb

The buffer size (in kilobytes) to use for reading contiguous columns. This should match the size of the columns typically retrieved using query operations involving a slice predicate.

stream_throughput_outbound_megabits_per_sec

Throttles all outbound streaming file transfers on a node to the specified throughput in Mb per second. Cassandra does mostly sequential I/O when streaming data during bootstrap or repair, which can lead to saturating the network connection and degrading client performance. The default is 400 Mb/s or 50 MB/s.

Remote Procedure Call Tuning Properties

The following properties are used to configure and tune remote procedure calls (client connections).

request_scheduler

Defines a scheduler to handle incoming client requests according to a defined policy. This scheduler only applies to client requests, not inter-node communication. Useful for throttling client requests in implementations that have multiple keyspaces. Allowed Values are:

- org.apache.cassandra.scheduler.NoScheduler (default)
- org.apache.cassandra.scheduler.RoundRobinScheduler
- A Java class that implements the RequestScheduler interface If using the `RoundRobinScheduler`, there are additional request_scheduler_options properties.

request_scheduler_id

An identifier on which to perform request scheduling. Currently the only valid option is `keyspace`.

request_scheduler_options

Contains a list of additional properties that define configuration options for request_scheduler. `NoScheduler` does not have any options. `RoundRobinScheduler` has the following additional configuration properties: throttle_limit, default_weight, weights.

throttle_limit

The number of active requests per client. Requests beyond this limit are queued up until running requests complete. The default is 80. Recommended value is ((concurrent_reads + concurrent_writes) * 2).

default_weight

The default weight controls how many requests are handled during each turn of the RoundRobin. The default is 1.

weights

Allows control of weight per keyspace during each turn of the RoundRobin. If not set, each keyspace uses the default_weight. Takes a list of list of `keyspaces: weights`.

rpc_keepalive

Enable or disable keepalive on client connections.

rpc_max_threads

Cassandra uses one thread-per-client for remote procedure calls. For a large number of client connections, this can cause excessive memory usage for the thread stack. Connection pooling on the client side is highly recommended. Setting a maximum thread pool size acts as a safeguard against misbehaved clients. If the maximum is reached, Cassandra will block additional connections until a client disconnects.

rpc_min_threads

Sets the minimum thread pool size for remote procedure calls.

rpc_recv_buff_size_in_bytes

Sets the receiving socket buffer size for remote procedure calls.

rpc_send_buff_size_in_bytes

Sets the sending socket buffer size in bytes for remote procedure calls.

rpc_timeout_in_ms

The time in milliseconds that a node will wait on a reply from other nodes before the command is failed.

rpc_server_type

Cassandra provides three options for the rpc server. The default is `sync` because `hsha` is about 30% slower on Windows. On Linux, `sync` and `hsha` performance is about the same with `hsha` using less memory.

- `sync` - (default) One connection per thread in the rpc pool. For a very large number of clients, memory will be your limiting factor; on a 64 bit JVM, 128KB is the minimum stack size per thread. Connection pooling is very, very strongly recommended.
- `hsha` Half synchronous, half asynchronous. The rpc thread pool is used to manage requests, but the threads are multiplexed across the different clients.
- `async` - Deprecated and will be removed in the next major release. Do not use.

thrift_framed_transport_size_in_mb

Specifies the frame size in megabytes (maximum field length) for Thrift. `0` disables framing. This option is deprecated in favor of thrift_max_message_length_in_mb.

thrift_max_message_length_in_mb

The maximum length of a Thrift message in megabytes, including all fields and internal Thrift overhead.

Internode Communication and Fault Detection Properties

dynamic_snitch

When set to true (default), enables the dynamic snitch layer that monitors read latency and, when possible, routes requests away from poorly-performing nodes.

dynamic_snitch_badness_threshold

Sets a performance threshold for dynamically routing requests away from a poorly performing node. A value of 0.2 means Cassandra would continue to prefer the static snitch values until the node response time was 20 percent worse than the best performing node.

Until the threshold is reached, incoming client requests are statically routed to the closest replica (as determined by the configured snitch). Having requests consistently routed to a given replica can help keep a working set of data *hot* when *read repair* is less than 100% or disabled.

dynamic_snitch_reset_interval_in_ms

Time interval in milliseconds to reset all node scores (allowing a bad node to recover).

dynamic_snitch_update_interval_in_ms

The time interval in milliseconds for calculating read latency.

hinted_handoff_enabled

Enables or disables hinted handoff.

hinted_handoff_throttle_delay_in_ms

When a node detects that a node for which it is holding hints has recovered, it begins sending the hints to that node. This specifies a sleep interval (in milliseconds) after delivering each row or row fragment in an effort to throttle traffic to the recovered node.

max_hint_window_in_ms

Defines how long in milliseconds to generate and save hints for an unresponsive node. After this interval, hints are dropped. This can prevent a sudden demand for resources when a node is brought back online and the rest of the cluster attempts to replay a large volume of hinted writes. The default is one hour (3600000 ms).

phi_convict_threshold

The Phi convict threshold adjusts the sensitivity of the failure detector on an exponential scale . Lower values increase the likelihood that an unresponsive node will be marked as down, while higher values decrease the likelihood that transient failures will cause a node failure. In unstable network environments (such as EC2 at times), raising the value to 10 or 12 will prevent false failures. Values higher than 12 and lower than 5 are not recommended. The default is 8.

Automatic Backup Properties

incremental_backups

Backs up data updated since the last snapshot was taken. When enabled, each time an SSTable is flushed, a hard link is copied into a `/backups` subdirectory of the keyspace data directory.

snapshot_before_compaction

Defines whether or not to take a snapshot before each compaction. Be careful using this option, since Cassandra does not clean up older snapshots automatically. This can be useful to back up data when there is a data format change.

Security Properties

authenticator

The default value disables authentication. Basic authentication is provided using the SimpleAuthenticator, which uses the `access.properties` and `password.properties` configuration files to configure authentication privileges. Allowed values are: * org.apache.cassandra.auth.AllowAllAuthenticator * org.apache.cassandra.auth.SimpleAuthenticator * A Java class that implements the IAuthenticator interface

Note

The `SimpleAuthenticator` and `SimpleAuthority` classes have been moved to the example directory of the Apache Cassandra project repository as of release 1.0. They are no longer available in the packaged and binary distributions. They never provided actual security, and in their current state are only meant as examples.

authority

The default value disables user access control (all users can access all resources). To control read/write permissions to keyspaces and column families, use the `SimpleAuthority`, which uses the `access.properties` configuration file to define per-user access. Allowed values are: * org.apache.cassandra.auth.AllowAllAuthority * org.apache.cassandra.auth.SimpleAuthority * A Java class that implements the IAuthority interface

internode_encryption

Enables or disables encryption of inter-node communication using `TLS_RSA_WITH_AES_128_CBC_SHA` as the cipher suite for authentication, key exchange and encryption of the actual data transfers. To encrypt all inter-node communications, set to `all`. You must also generate keys and provide the appropriate key and trust store locations and passwords.

keystore

Description: The location of a Java keystore (JKS) suitable for use with Java Secure Socket Extension (JSSE), the Java version of the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. The keystore contains the private key used to encrypt outgoing messages.

keystore_password

Password for the keystore.

truststore

The location of a truststore containing the trusted certificate used to authenticate remote servers.

truststore_password

Password for the truststore.

Keyspace and Column Family Storage Configuration

Many aspects of storage configuration are set on a per-keyspace or per-column family basis. These attributes can be manipulated programmatically, but in most cases the practical method for defining keyspace and column family attributes is to use the Cassandra CLI or CQL interfaces.

Prior to release 0.7.3, keyspace and column family attributes could be specified in `cassandra.yaml`, but that is no longer true in 0.7.4 and later. These attributes are now stored in the `system` keyspace within Cassandra.

Note

The attribute names documented in this section are the names as they are stored in the `system` keyspace within Cassandra. Most of these attributes can be set in the various client applications, such as Cassandra CLI or CQL. There may be slight differences in how these attributes are named depending on how they are implemented in the client.

Keyspace Attributes

A keyspace must have a user-defined name and a replica placement strategy. It also has replication strategy options, which is a container attribute for replication factor or the number of replicas per data center.

Option	Default Value
ks_name	n/a (A user-defined value is required)
placement_strategy	org.apache.cassandra.locator.SimpleStrategy
strategy_options	n/a (container attribute)

name

Required. The name for the keyspace.

placement_strategy

Required. Determines how replicas for a keyspace will be distributed among nodes in the ring.

Allowed values are:

- `org.apache.cassandra.locator.SimpleStrategy`
- `org.apache.cassandra.locator.NetworkTopologyStrategy`
- `org.apache.cassandra.locator.OldNetworkTopologyStrategy` (deprecated)

These options are described in detail in the *replication* section.

Note

`NetworkTopologyStrategy` and `OldNetworkTopologyStrategy` require a properly configured snitch to be able to determine rack and data center locations of a node (see *endpoint_snitch*).

strategy_options

Specifies configuration options for the chosen replication strategy.

For `SimpleStrategy`, it specifies `replication_factor` in the format of `replication_factor`:*number_of_replicas*.

For `NetworkTopologyStrategy`, it specifies the number of replicas per data center in a comma separated list of *datacenter_name*:*number_of_replicas*. Note that what you specify for *datacenter_name* depends on the cluster-configured *snitch* you are using. There is a correlation between the data center name defined in the keyspace `strategy_options` and the data center name as recognized by the snitch you are using. The *nodetool ring* command prints out data center names and rack locations of your nodes if you are not sure what they are.

See *Choosing Keyspace Replication Options* for guidance on how to best configure replication strategy and strategy options for your cluster.

Setting and updating strategy options with the Cassandra CLI requires a slightly different command syntax than other attributes; note the brackets and curly braces in this example:

```
[default@unknown] CREATE KEYSPACE test
WITH placement_strategy = 'NetworkTopologyStrategy'
AND strategy_options=[{us-east:6,us-west:3}];
```

Column Family Attributes

The following attributes can be declared per column family.

Option	Default Value
column_metadata	n/a (container attribute)
column_type	Standard
comment	n/a
compaction_strategy	SizeTieredCompactionStrategy
compaction_strategy_options	n/a (container attribute)
comparator	BytesType
compare_subcolumns_with	BytesType
compression_options	n/a (container attribute)
default_validation_class	n/a
gc_grace_seconds	864000 (10 days)
key_validation_class	n/a
key_cache_save_period_in_seconds	n/a
keys_cached	200000
max_compaction_threshold	32
min_compaction_threshold	4
memtable_flush_after_mins	ignored in 1.0 and later releases
memtable_operations_in_millions	ignored in 1.0 and later releases
memtable_throughput_in_mb	ignored in 1.0 and later releases
cf_name	n/a (A user-defined value is required)
read_repair_chance	0.1 (repair 10% of the time)
replicate_on_write	true
rows_cached	0 (disabled by default)
row_cache_provider	ConcurrentLinkedHashCacheProvider
row_cache_save_period_in_seconds	n/a

column_metadata

Column metadata defines attributes of a column. Values for `name` and `validation_class` are required, though the default_validation_class for the column family is used if no validation_class is specified. Note that `index_type` must be set to create a secondary index for a column. `index_name` is not valid unless `index_type` is also set.

Name	Description
name	Binds a validation_class and (optionally) an index to a column.
validation_class	Type used to check the column value.
index_name	Name for the secondary index.
index_type	Type of index. Currently the only supported value is KEYS.

Setting and updating column metadata with the Cassandra CLI requires a slightly different command syntax than other attributes; note the brackets and curly braces in this example:

```
[default@demo] UPDATE COLUMN FAMILY users WITH comparator=UTF8Type
AND column_metadata=[{column_name: full_name, validation_class: UTF8Type, index_type: KEYS}];
```

column_type

Defaults to `Standard` for regular column families. For super column families, use `Super`.

comment

A human readable comment describing the column family.

compaction_strategy

Sets the compaction strategy for the column family. The available strategies are:

- SizeTieredCompactionStrategy - This is the default compaction strategy and the only compaction strategy available in pre-1.0 releases. This strategy triggers a minor compaction whenever there are a number of similar sized SSTables on disk (as configured by min_compaction_threshold). This strategy causes bursts in I/O activity while a compaction is in process, followed by longer and longer lulls in compaction activity as SSTable files grow larger in size. These I/O bursts can negatively effect read-heavy workloads, but typically do not impact write performance. Watching disk capacity is also important when using this strategy, as compactions can temporarily double the size of SSTables for a column family while a compaction is in progress.
- LeveledCompactionStrategy - The leveled compaction strategy limits the size of SSTables to a small file size (5 MB by default) so that SSTables do not continue to grow in size with each successive compaction. Disk I/O is more uniform and predictable as SSTables are continuously being compacted into progressively larger levels. At each level, row keys are merged into non-overlapping SSTables. This can improve performance for reads, because Cassandra can determine which SSTables in each level to check for the existence of row key data. This compaction strategy is modeled after Google's leveldb implementation.

compaction_strategy_options

Sets options related to the chosen compaction_strategy. Currently only `LeveledCompactionStrategy` has options.

Option	Default Value	Description
sstable_size_in_mb	5	Sets the file size for leveled SSTables. A compaction is triggered when unleveled SSTables (newly flushed SSTable files in Level 0) exceeds `4 * sstable_size_in_mb`.

Setting and updating compaction strategy options with the Cassandra CLI requires a slightly different command syntax than other attributes; note the brackets and curly braces in this example:

```
[default@demo] UPDATE COLUMN FAMILY users WITH compaction_strategy=LeveledCompactionStrategy
AND compaction_strategy_options=[{sstable_size_in_mb: 10}];
```

comparator

Defines the data types used to validate and sort column names. There are several built-in *column comparators* available. Note that the comparator cannot be changed after a column family is created.

compare_subcolumns_with

Required when column_type is "Super". Same as comparator but for sub-columns of a SuperColumn.

For attributes of columns, see column_metadata.

compression_options

This is a container attribute for setting compression options on a column family. It contains the following options:

Option	Description
sstable_compression	Specifies the compression algorithm to use when compressing SSTable files. Cassandra supports two built-in compression classes: `SnappyCompressor` (Snappy compression library) and `DeflateCompressor` (Java zip implementation). Snappy compression offers faster compression/decompression while the Java zip compression offers better compression ratios. Choosing the right one depends on your requirements for space savings over read performance. For read-heavy workloads, Snappy compression is recommended. Developers can also implement custom compression classes using the `org.apache.cassandra.io.compress.ICompressor` interface.
chunk_length_kb	Sets the compression chunk size in kilobytes. The default value (64) is a good middle-ground for compressing column families with either wide rows or with skinny rows. With wide rows, it allows reading a 64kb slice of column data without decompressing the entire row. For skinny rows, although you may still end up decompressing more data than requested, it is a good trade-off between maximizing the compression ratio and minimizing the overhead of decompressing more data than is needed to access a requested row.The compression chunk size can be adjusted to account for read/write access patterns (how much data is typically requested at once) and the average size of rows in the column family.

Setting and updating compression options with the Cassandra CLI requires a slightly different command syntax than other attributes; note the brackets and curly braces in this example:

```
[default@demo] UPDATE COLUMN FAMILY users WITH compression_options=[{sstable_compression:SnappyCompressor, chunk_length_kb:64}];
```

default_validation_class

Defines the data type used to validate column values. There are several built-in *column validators* available.

gc_grace_seconds

Specifies the time to wait before garbage collecting tombstones (deletion markers). Defaults to 864000, or 10 days, which allows a great deal of time for consistency to be achieved prior to deletion. In many deployments this interval can be reduced, and in a single-node cluster it can be safely set to zero.

Note

This property is called `gc_grace` in the `cassandra-cli` client.

key_cache_save_period_in_seconds

Sets number of seconds between saving key caches: the key caches can be saved periodically, and if one exists on startup it will be loaded.

Note

This property is called `key_cache_save_period` in the `cassandra-cli` client.

keys_cached

Defines how many key locations will be kept in memory per SSTable (see rows_cached for details on caching actual row values). This can be a fixed number of keys or a fraction (for example 0.5 means 50 percent).

DataStax recommends a fixed sized cache over a relative sized cache. Only use relative cache sizing when you are confident that the data in the column family will not continue to grow over time. Otherwise, your cache will grow as your data set does, potentially causing unplanned memory pressure.

key_validation_class

Defines the data type used to validate row key values. There are several built-in *key validators* available, however `CounterColumnType` (distributed counters) cannot be used as a row key validator.

name

Required. The user-defined name of the column family.

read_repair_chance

Specifies the probability with which read repairs should be invoked on non-quorum reads. Must be between 0 and 1. Defaults to 0.1 (perform read repair 10% of the time). Lower values improve read throughput, but increase the chances of seeing stale values if you are not using a strong *consistency level*.

replicate_on_write

When set to `true`, replicates writes to all affected replicas regardless of the consistency level specified by the client for a write request. For counter column families, this should always be set to `true`.

max_compaction_threshold

Sets the maximum number of SSTables to allow in a minor compaction when `compaction_strategy=SizeTieredCompactionStrategy`. Obsolete as of Cassandra 0.8 with the addition of compaction throttling (see `cassandra.yaml` parameter *compaction_throughput_mb_per_sec*).

Setting this to `0` disables minor compactions. Defaults to 32.

min_compaction_threshold

Sets the minimum number of SSTables to trigger a minor compaction when `compaction_strategy=sizeTieredCompactionStrategy`. Raising this value causes minor compactions to start less frequently and be more I/O-intensive. Setting this to 0 disables minor compactions. Defaults to 4.

memtable_flush_after_mins

Deprecated as of Cassandra 1.0. Can still be declared (for backwards compatibility) but settings will be ignored. Use the `cassandra.yaml` parameter *commitlog_total_space_in_mb* instead.

memtable_operations_in_millions

Deprecated as of Cassandra 1.0. Can still be declared (for backwards compatibility) but settings will be ignored. Use the `cassandra.yaml` parameter *commitlog_total_space_in_mb* instead.

memtable_throughput_in_mb

Deprecated as of Cassandra 1.0. Can still be declared (for backwards compatibility) but settings will be ignored. Use the `cassandra.yaml` parameter *commitlog_total_space_in_mb* instead.

rows_cached

Specifies how many rows to cache in memory. This can be a fixed number of rows or a fraction (for example 0.5 means 50 percent).

Using a row cache means that the entire row is cached in memory. This can be detrimental to performance in cases where rows are large, or where rows are frequently modified or removed.

row_cache_provider

Specifies the row cache to use for the column family. Allowed values are: * `ConcurrentLinkedHashCacheProvider` - Rows are cached using the JVM heap, providing the same row cache behavior as Cassandra versions prior to 0.8. * (default) `SerializingCacheProvider` - Cached rows are serialized and stored in memory off of the JVM heap, which can reduce garbage collection (GC) pressure on the JVM and thereby improve system performance. Serialized rows are also 8-12 times smaller than unserialized rows. This is the recommended setting, as long as you have `jna.jar` in the CLASSPATH to enable native methods.

row_cache_save_period_in_seconds

Sets the number of seconds between saving row caches: the row caches can be saved periodically, and if one exists on startup it will be loaded.

Java and System Environment Settings Configuration

There are two files that control environment settings for Cassandra:

- `conf/cassandra-env.sh` - Java Virtual Machine (JVM) configuration settings
- `bin/cassandra-in.sh` - Sets up Cassandra environment variables such as `CLASSPATH` and `JAVA_HOME`.

Heap Sizing Options

If you decide to change the Java heap sizing, both MAX_HEAP_SIZE and HEAP_NEWSIZE should should be set together in `conf/cassandra-env.sh` (if you set one, set the other as well). See the section on *Tuning Java Heap Size* for more information on choosing the right Java heap size.

- `MAX_HEAP_SIZE` - Sets the maximum heap size for the JVM. The same value is also used for the minimum heap size. This allows the heap to be locked in memory at process start to keep it from being swapped out by the OS. Defaults to half of available physical memory.
- `HEAP_NEWSIZE` - The size of the young generation. The larger this is, the longer GC pause times will be. The shorter it is, the more expensive GC will be (usually). A good guideline is 100 MB per CPU core.

JMX Options

Cassandra exposes a number of statistics and management operations via Java Management Extensions (JMX). Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring Java applications and services. Any statistic or operation that a Java application has exposed as an MBean can then be monitored or manipulated using JMX. JConsole, *nodetool* and DataStax OpsCenter are examples of JMX-compliant management tools.

By default, the `conf/cassandra-env.sh` file configures JMX to listen on port 7199 without authentication. See the table below for more information on commonly changed JMX configuration properties.

- `com.sun.management.jmxremote.port` - The port on which Cassandra listens from JMX connections
- `com.sun.management.jmxremote.ssl` - Enable/disable SSL for JMX
- `com.sun.management.jmxremote.authenticate` - Enable/disable remote authentication for JMX
- `-Djava.rmi.server.hostname` - Sets the interface hostname or IP that JMX should use to connect. Uncomment and set if you are having trouble connecting.

Further Reading on JVM Tuning

The remaining options are optimal across a wide variety of workloads and environments and are not frequently changed. See the Sun JVM options list for more information on JVM tuning parameters.

Authentication and Authorization Configuration

Note

The `SimpleAuthenticator` and `SimpleAuthority` classes have been moved to the example directory of the Apache Cassandra project repository as of release 1.0. They are no longer available in the packaged and binary distributions. They never provided actual security, and in their current state are only meant as examples.

Using authentication and authorization requires configuration changes in `cassandra.yaml` and two additional files: one for assigning users and their permissions to keyspaces and column families, and the other for assigning passwords to those users. These files are named `access.properties` and `passwd.properties`, respectively, and are located in the `examples` directory of the Apache Cassandra project repository. To test simple authentication, you can move these files to the `conf` directory.

To set up simple authentication and authorization

1. Edit *cassandra.yaml*, setting `org.apache.cassandra.auth.SimpleAuthenticator` as the `authenticator` value. The default value of `AllowAllAuthenticator` is equivalent to no authentication.
2. Edit `access.properties`, adding entries for users and their permissions to read and write to specified keyspaces and column families. See *access.properties* below for details on the correct format.
3. Make sure that users specified in `access.properties` have corresponding entries in `passwd.properties`. See *passwd.properties* below for details and examples.
4. After making the required configuration changes, you must specify the properties files when starting Cassandra with the flags `-Dpasswd.properties` and `-Daccess.properties.` For example:

```
cd $CASSANDRA_HOME
sh bin/cassandra -f -Dpasswd.properties=conf/passwd.properties -Daccess.properties=conf/access.properties
```

access.properties

This file contains entries in the format `KEYSPACE[.COLUMNFAMILY].PERMISSION=USERS` where

- KEYSPACE is the keyspace name.
- COLUMNFAMILY is the column family name.
- PERMISSION is one of <ro> or <rw> for read-only or read-write respectively.
- USERS is a comma delimited list of users from `passwd.properties`.

For example, to control access to Keyspace1 and give jsmith and Elvis read-only permissions while allowing dilbert full read-write access to add and remove column families, you would create the following entries:

```
Keyspace1.<ro>=jsmith,Elvis Presley
Keyspace1.<rw>=dilbert
```

To provide a finer level of access control to the Standard1 column family in Keyspace1, you would create the following entry to allow the specified users read-write access:

```
Keyspace1.Standard1.<rw>=jsmith,Elvis Presley,dilbert
```

The `access.properties` file also contains a simple list of users who have permissions to modify the list of keyspaces:

```
<modify-keyspaces>=jsmith
```

passwd.properties

This file contains name/value pairs in which the names match users defined in `access.properties` and the values are user passwords. Passwords are in clear text unless the `passwd.mode=MD5` system property is provided.

```
jsmith=havebadpass
Elvis Presley=graceland4ever
dilbert=nomoovertime
```

Logging Configuration

In some situations, the output provided by Cassandra's JMX MBeans and the *nodetool* utility are not enough to diagnose issues. If you find yourself in a place where you need more information about the runtime behavior of a specific Cassandra node, you can increase the logging levels to get more diagnostic information on specific portions of the system.

Cassandra uses a SLF4J to provide logging functionality; by default a log4j backend is used. Many of the core Cassandra classes have varying levels of logging output available which can be increased in one of two ways:

1. Updating the `log4j-server.properties` file
2. Through JMX

Logging Levels via the Properties File

To update the via properties file, edit `conf/log4j-server.properties` to include the following two lines (warning - this will generate *a lot* of logging output on even a moderately trafficked cluster):

```
log4j.logger.org.apache.cassandra.db=DEBUG
log4j.logger.org.apache.cassandra.service.StorageProxy=DEBUG
```

This will apply the `DEBUG` log level to all the classes in `org.apache.cassandra.db` package and below as well as to the StorageProxy class directly. Cassandra checks the log4j configuration file every ten seconds, and applies changes without needing a restart.

Note that by default, logging will go to `/var/log/cassandra/system.log`; the location of this log file can be changed as well - just update the `log4j.appender.R.File` path to where you would like the log file to exist, and ensure that the directory exists and is writable by the process running Cassandra.

Additionally, the default configuration will roll the log file once the size exceeds 20MB and will keep up to 50 backups. These values may be changed as well to meet your requirements.

Logging Levels via JMX

To change logging levels via JMX, bring up the JConsole tool and attach it to the CassandraDaemon process. Locate the StorageService MBean and find `setLog4jLevel` under the Operations list.

This operation takes two arguments - a class qualifier and a logging level. The class qualifier can either be a full class name or any portion of a package name, similar to the `log4j-server.properties` configuration above except without the initial `log4j.logger` appender assignment. The level must be one of the standard logging levels in use by Log4j.

In keeping with our initial example, to adjust the logging output of StorageProxy to `DEBUG`, the first argument would be `org.apache.cassandra.service.StorageProxy`, and the second one `DEBUG`. On a system with traffic, you should see the effects of this change immediately.

Operations

Monitoring a Cassandra Cluster

Understanding the performance characteristics of your Cassandra cluster is critical to diagnosing issues and planning capacity.

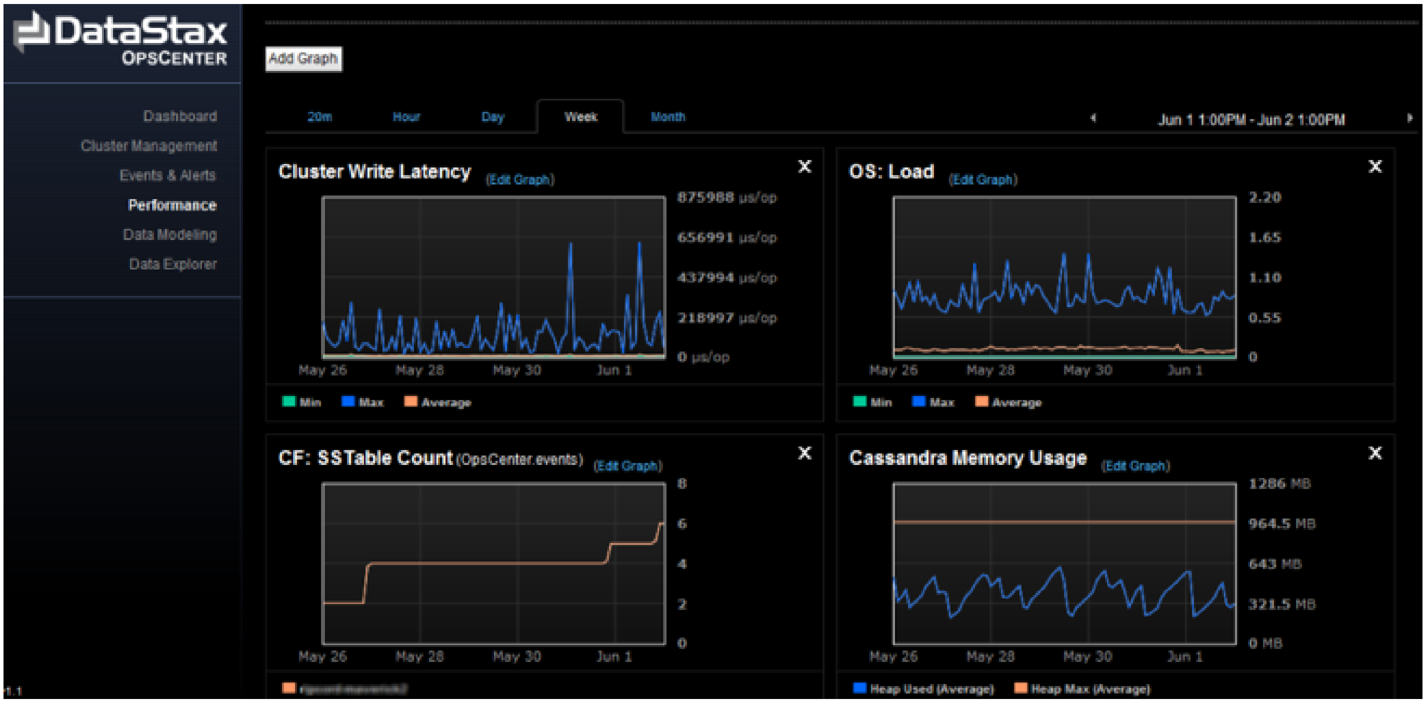
Cassandra exposes a number of statistics and management operations via Java Management Extensions (JMX). Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring Java applications and services. Any statistic or operation that a Java application has exposed as an MBean can then be monitored or manipulated using JMX.

During normal operation, Cassandra outputs information and statistics that you can monitor using JMX-compliant tools such as JConsole, the Cassandra *nodetool* utility, or the DataStax OpsCenter management console. With the same tools, you can perform certain administrative commands and operations such as flushing caches or doing a repair.

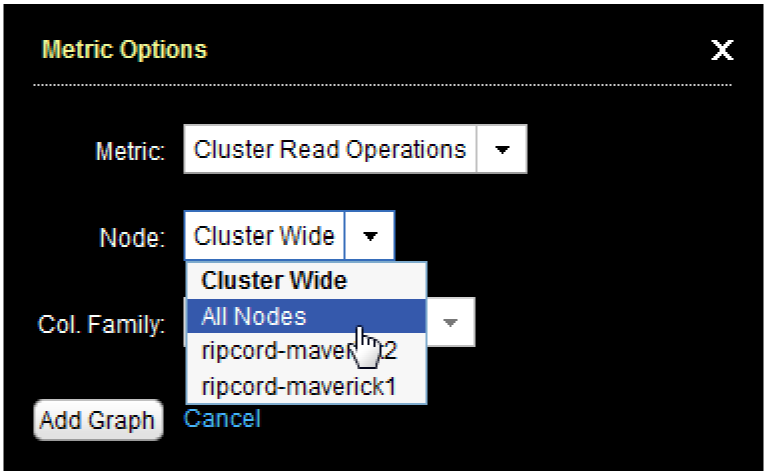
Monitoring Using DataStax OpsCenter

DataStax OpsCenter is a graphical user interface for monitoring and administering all nodes in a Cassandra cluster from one centralized console. DataStax OpsCenter is bundled with DataStax support offerings, or you can register for a free version licensed for development or non-production use.

OpsCenter provides a graphical representation of performance trends in a summary view that is hard to obtain with other monitoring tools. The GUI provides views for different time periods as well as the capability to drill down on single data points. Both real-time and historical performance data for a Cassandra or Brisk cluster are available in OpsCenter. OpsCenter metrics are captured and stored within Cassandra.



The performance metrics viewed within OpsCenter can be customized according to your monitoring needs. Administrators can also perform routine node administration tasks from OpsCenter. Metrics within OpsCenter are divided into three general categories: column family metrics, cluster metrics, and OS metrics. For many of the available metrics, you can choose to view aggregated cluster-wide information, or view information on a per-node basis.


Metric Options

Metric: Cluster Read Operations

Node: Cluster Wide

Col. Family:

Monitoring Using `nodetool`

The *nodetool* utility is a command-line interface for monitoring Cassandra and performing routine database operations. It is included in the Cassandra distribution and is typically run directly from an operational Cassandra node.

The `nodetool` utility supports the most important JMX metrics and operations, and includes other useful commands for Cassandra administration. This utility is commonly used to output a quick summary of the ring and its current state of general health with the `ring` command. For example:

```
# nodetool -h localhost -p 7199 ring
Address         Status   State    Load         Owns     Range                                        Ring
                                                        95315431979199388464207182617231204396
```

```
10.194.171.160 Down     Normal  ?           39.98   61078635599166706937511052402724559481     |<--|
10.196.14.48   Up       Normal  3.16 KB     30.01   78197033789183047700859117509977881938     |   |
10.196.14.239  Up       Normal  3.16 KB     30.01   95315431979199388464207182617231204396     |-->|
```

The nodetool utility provides commands for viewing detailed metrics for column family metrics, server metrics, and compaction statistics. Commands are also available for important operations such as decommissioning a node, running repair, and moving partitioning tokens.

Monitoring Using JConsole

JConsole is a JMX-compliant tool for monitoring Java applications such as Cassandra. It is included with Sun JDK 5.0 and higher. JConsole consumes the JMX metrics and operations exposed by Cassandra and displays them in a well-organized GUI. For each node monitored, JConsole provides these six separate tab views:

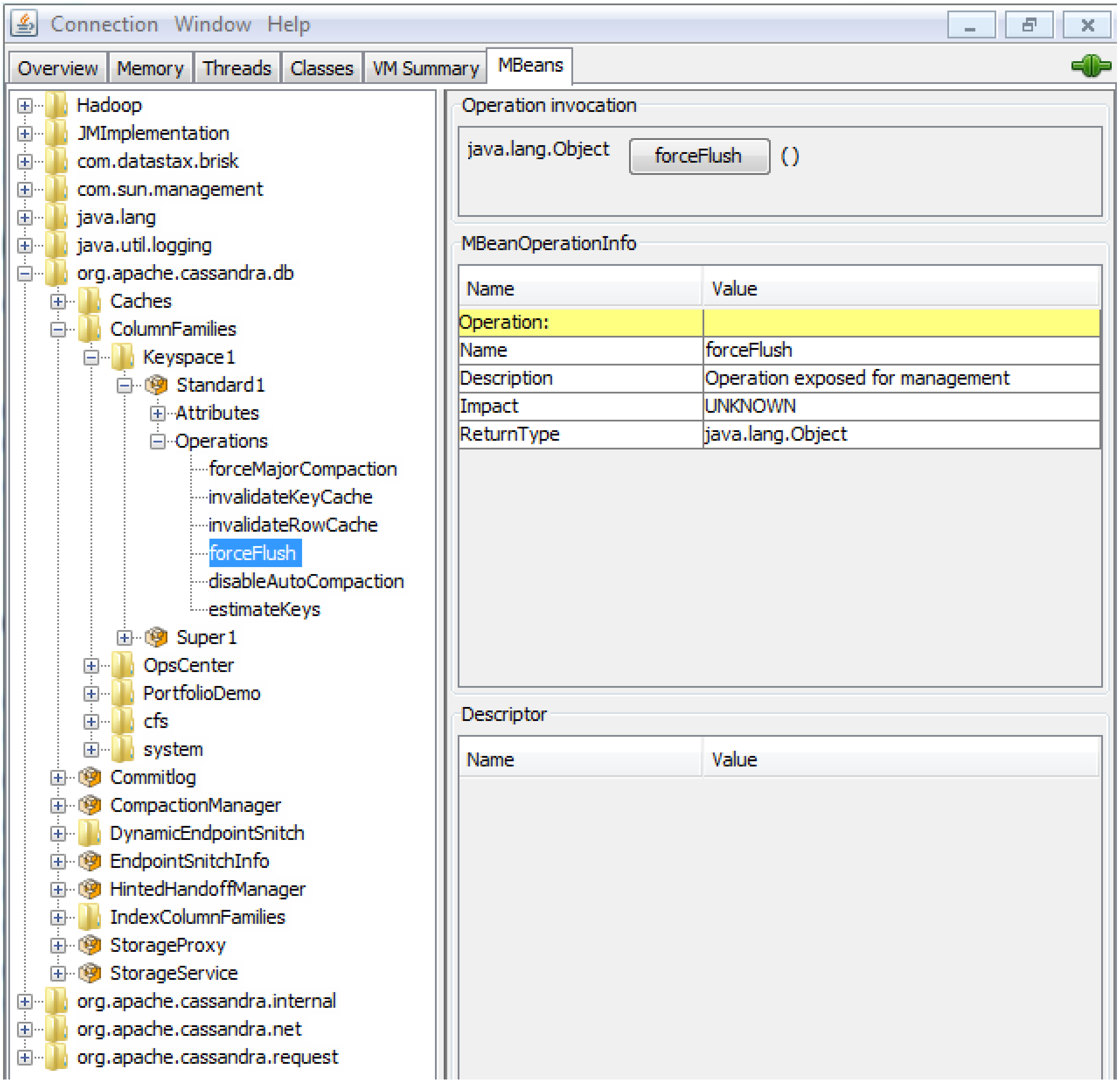
- **Overview** - Displays overview information about the Java VM and monitored values.
- **Memory** - Displays information about memory use.Threads - Displays information about thread use.
- **Classes** - Displays information about class loading.
- **VM Summary** - Displays information about the Java Virtual Machine (VM).
- **Mbeans** - Displays information about MBeans.

The **Overview** and **Memory** tabs contain information that is very useful for Cassandra developers. The Memory tab allows you to compare heap and non-heap memory usage, and provides a control to immediately perform Java garbage collection.

For specific Cassandra metrics and operations, the most important area of JConsole is the **MBeans** tab. This tab lists the following Cassandra MBeans:

- `org.apache.cassandra.db` - Includes caching, column family metrics, and compaction.
- `org.apache.cassandra.internal` - Internal server operations such as gossip and hinted handoff.
- `org.apache.cassandra.net` - Inter-node communication including FailureDetector, MessagingService and StreamingService.
- `org.apache.cassandra.request` - Tasks related to read, write, and replication operations.

When you select an MBean in the tree, its MBeanInfo and MBean Descriptor are both displayed on the right, and any attributes, operations or notifications appear in the tree below it. For example, selecting and expanding the `org.apache.cassandra.db` MBean to view available actions for a column family results in a display like the following:



Name	Value
Operation:	
Name	forceFlush
Description	Operation exposed for management
Impact	UNKNOWN
ReturnType	java.lang.Object

Name	Value

If you choose to monitor Cassandra using JConsole, keep in mind that JConsole consumes a significant amount of system resources. For this reason, DataStax recommends running JConsole on a remote machine rather than on the same host as a Cassandra node.

Compaction Metrics

Monitoring compaction performance is an important aspect of knowing when to add capacity to your cluster. The following attributes are exposed through `CompactionManagerMBean`:

Attribute	Description
CompletedTasks	Number of completed compactions since the last start of this Cassandra instance
PendingTasks	Number of estimated tasks remaining to perform
ColumnFamilyInProgress	ColumnFamily currently being compacted. `null` if no compactions are in progress.
BytesTotalInProgress	Total number of data bytes (index and filter are not included) being compacted. `null` if no compactions are in progress.

BytesCompacted	The progress of the current compaction. `null` if no compactions are in progress.

Thread Pool Statistics

Cassandra maintains distinct thread pools for different stages of execution. Each of these thread pools provide statistics on the number of tasks that are active, pending and completed. Watching trends on these pools for increases in the pending tasks column is an excellent indicator of the need to add additional capacity. Once a baseline is established, alarms should be configured for any increases past normal in the pending tasks column. See below for details on each thread pool (this list can also be obtained via command line using *nodetool tpstats*).

Thread Pool	Description
AE_SERVICE_STAGE	Shows anti-entropy tasks
CONSISTENCY-MANAGER	Handles the background consistency checks if they were triggered from the client's *consistency level <consistency>*
FLUSH-SORTER-POOL	Sorts flushes that have been submitted
FLUSH-WRITER-POOL	Writes the sorted flushes
GOSSIP_STAGE	Activity of the Gossip protocol on the ring
LB-OPERATIONS	The number of load balancing operations
LB-TARGET	Used by nodes leaving the ring
MEMTABLE-POST-FLUSHER	Memtable flushes that are waiting to be written to the commit log.
MESSAGE-STREAMING-POOL	Streaming operations. Usually triggered by bootstrapping or decommissioning nodes.
MIGRATION_STAGE	Tasks resulting from the call of `system_*` methods in the API that have modified the schema
MISC_STAGE	
MUTATION_STAGE	API calls that are modifying data
READ_STAGE	API calls that have read data
RESPONSE_STAGE	Response tasks from other nodes to message streaming from this node
STREAM_STAGE	Stream tasks from this node

Read/Write Latency Metrics

Cassandra keeps tracks latency (averages and totals) of read, write and slicing operations at the server level through StorageProxyMBean.

ColumnFamily Statistics

For individual column families, `ColumnFamilyStoreMBean` provides the same general latency attributes as `StorageProxyMBean`. Unlike `StorageProxyMBean`, `ColumnFamilyStoreMBean` has a number of other statistics that are important to monitor for performance trends. The most important of these are listed below:

Attribute	Description
MemtableDataSize	The total size consumed by this column family's data (not including meta data)
MemtableColumnsCount	Returns the total number of columns present in the memtable (across all keys)
MemtableSwitchCount	How many times the memtable has been flushed out
RecentReadLatencyMicros	The average read latency since the last call to this bean
RecentWriterLatencyMicros	The average write latency since the last call to this bean

LiveSSTableCount	The number of live SSTables for this ColumnFamily

The first three Memtable attributes are discussed in detail on the *Tuning Cassandra* page.

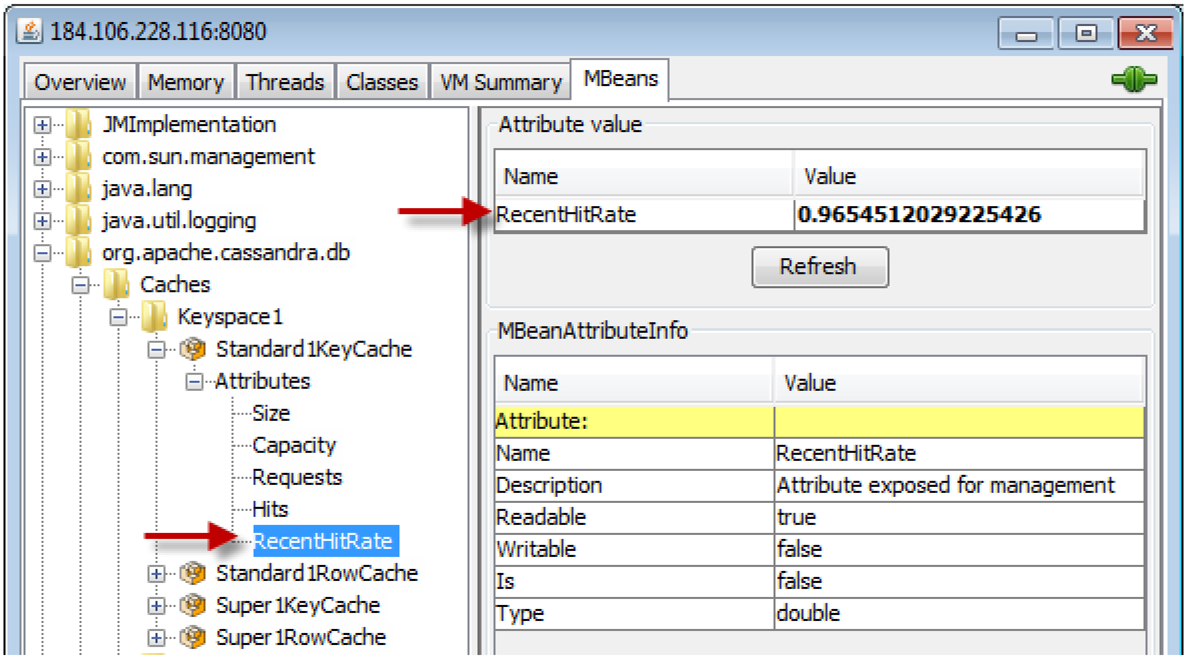
The recent read latency and write latency counters are important in making sure that operations are happening in a consistent manner. If these counters start to increase after a period of staying flat, it is probably an indication of a need to add cluster capacity.

`LiveSSTableCount` can be monitored with a threshold to ensure that the number of SSTables for a given ColumnFamily does not become too great.

Monitoring and Adjusting Cache Performance

Careful, incremental monitoring of cache changes is the best way to maximize benefit from Cassandra's built-in caching features. Adjustments that increase cache hit rate are likely to use more system resources, such as memory. After making changes to the cache configuration, it is best to monitor Cassandra as a whole for unintended impact on the system.

For each node and each column family, you can view cache hit rate, cache size, and number of hits by expanding `org.apache.cassandra.db` in the MBeans tab. For example:



Name	Value
RecentHitRate	0.9654512029225426

Name	Value
Attribute:	
Name	RecentHitRate
Description	Attribute exposed for management
Readable	true
Writable	false
Is	false
Type	double

Monitor new cache settings not only for hit rate, but also to make sure that memtables and heap size still have sufficient memory for other operations. If you cannot maintain the desired key cache hit rate of 85% or better, add nodes to the system and re-test until you can meet your caching requirements.

Row cache is disabled by default. Caching large rows can very quickly consume memory. Row cache rates should be increased carefully in small increments. If row cache hit rates cannot be tuned to above 30%, it may make more sense to leave row caching disabled.

Tuning Cassandra

Effective tuning depends not only on the types of operations your cluster performs most frequently, but also on the shape of the data itself. For example, Cassandra's memtables have overhead for index structures on top of the actual data they store. If the size of the values stored in the columns is small compared to the number of columns and rows themselves (sometimes called *skinny rows*), this overhead can be substantial. Thus, the optimal tuning for this type of data is quite different than the optimal tuning for a small numbers of columns with more data (*fat rows*).

Tuning the Cache

Cassandra's built-in key and row caches can provide very efficient data caching. Some Cassandra production deployments have leveraged Cassandra's caching features to the point where dedicated caching tools such as memcached could be completely replaced. Such deployments not only remove a redundant layer from the stack, but they also achieve the fundamental efficiency of strengthening caching functionality in the lower tier where the data is already being stored. Among other advantages, this means that caching never needs to be restarted in a completely cold state.

Cache tuning should be done using small, incremental adjustments and then monitoring the effects of each change. See *Monitoring and Adjusting Cache Performance* for more information about monitoring tuning changes to a column family cache. With proper tuning, key cache hit rates of 85% or better are possible. Row caching, when feasible, can save the system from performing any disk seeks at all when fetching a cached row. Whenever growth in the read load begins to impact your cache hit rates, you can add capacity to quickly restore optimal levels of caching.

How Caching Works

If both row and key caches are configured, the row cache will return results whenever possible. In the case of a row cache miss, the key cache may still provide a hit, assuming that it holds a larger number of keys than the row cache.

If a read operation hits the row cache, the entire requested row is returned without a disk seek. If a row is not in the row cache, but is present in the key cache, the key cache is used to find the exact location of the row on disk in the SSTable. If a row is not in the key cache, the read operation will populate the key cache after accessing the row on disk so subsequent reads of the row can benefit. Each hit on a key cache can save one disk seek per SSTable.

Configuring the Column Family Key Cache

The key cache holds the location of row keys in memory on a per-column family basis. High levels of key caching are recommended for most production scenarios. Turning this level up can optimize reads (after the cache warms) when there is a large number of rows that are accessed frequently.

The caching of 200,000 row keys is enabled by default. This can be adjusted by setting *keys_cached* on a column family. For example, using Cassandra CLI:

```
[default@demo] UPDATE COLUMN FAMILY users WITH keys_cached=205000;
```

Key cache performance can be monitored by using *nodetool cfstats* and examining the reported 'Key cache hit rate'. See also *Monitoring and Adjusting Cache Performance* for more information about monitoring tuning changes to a column family key cache.

Configuring the Column Family Row Cache

The row cache holds the entire contents of the row in memory. In cases where rows are large or frequently modified/removed, row caching can actually be detrimental to performance. For this reason, row caching is disabled by default.

Row cache should remain disabled for column families with large rows or high write:read ratios. In these situations, row cache can very quickly consume a large amount of available memory. Note also that, when a row cache is operating efficiently, it keeps Java garbage compaction processes very active.

Row caching is best for workloads that access a small subset of the overall rows, and within those rows, all or most of the columns are returned. For this use case a row cache keeps the most accessed rows hot in memory, and can have substantial performance benefits.

To enable row cache on a column family, set *rows_cached* to the desired number of rows. To enable off-heap row caching, set *row_cache_provider* to `SerializingCacheProvider`. For example, using Cassandra CLI:

```
[default@demo] UPDATE COLUMN FAMILY users WITH rows_cached=100000
AND row_cache_provider='SerializingCacheProvider';
```

Row cache performance can be monitored by using *nodetool cfstats* and examining the reported 'Row cache hit rate'. See also *Monitoring and Adjusting Cache Performance* for more information about monitoring tuning changes to a column family key cache.

Data Modeling Considerations for Cache Tuning

If your requirements permit it, a data model that logically separates heavily-read data into discrete column families can help optimize caching. Column families with relatively small, narrow rows lend themselves to highly efficient row caching. By the same token, it can make sense to separately store lower-demand data, or data with extremely long rows, in a column family with minimal caching, if any.

Row caching in such contexts brings the most benefit when access patterns follow a normal (Gaussian) distribution. When the keys most frequently requested follow such patterns, cache hit rates tend to increase. If you have particularly hot rows in your data model, row caching can bring significant performance improvements.

Hardware and OS Considerations for Cache Tuning

Deploying a large number of Cassandra nodes under a relatively light load per node will maximize the fundamental benefit from key and row caches.

A less obvious but very important consideration is the OS page cache. Modern operating systems maintain page caches for frequently accessed data and are very efficient at keeping this data in memory. Even after a row is released in the Java Virtual Machine memory, it can be kept in the OS page cache -- especially if the data is requested repeatedly, or no other requested data replaces it.

If your requirements allow you to lower *JVM heap size* and *memtable sizes* to leave memory for OS page caching, then do so. Ultimately, through gradual adjustments, you should achieve the desired balance between these three demands on available memory: heap, memtables, and caching.

Estimating Cache Sizes

nodetool cfstats can be used to get the necessary information for estimating actual cache sizes.

To estimate the key cache size, multiply the reported 'Key cache size' for each column family by 10-12 to account for the difference in reported Java heap usage and actual size. Then, add the results for all column families.

To estimate the row cache size, multiply the reported 'Row cache size' for each column family (which is the number of rows in the cache) by the average size of each row. Then multiply that figure by 10-12 and sum the results over all column families. As of Cassandra 1.0, the row cache for a column family is stored in native memory by default rather than using the JVM heap (see *row_cache_provider*).

Tuning Write Performance (Memtables)

A memtable is a column family specific, in memory data structure that can be easily described as a write-back cache. Memtables are flushed to disk, creating SSTables whenever one of the configurable thresholds has been exceeded.

Effectively tuning memtable thresholds depends on your data as much as your write load. Memtable thresholds are configured per node using the `cassandra.yaml` properties: *memtable_throughput_in_mb* and *commitlog_total_space_in_mb*.

You should increase memtable throughput if:

1. Your write load includes a high volume of updates on a smaller set of data
2. You have steady stream of continuous writes (this will lead to more efficient compaction)

Note that increasing memory allocations for memtables takes memory away from caching and other internal Cassandra structures, so tune carefully and in small increments.

Tuning Java Heap Size

Cassandra's default configuration opens the JVM with a heap size of 1/4 of the available system memory (or a minimum 1GB and maximum of 8GB for systems with a very low or very high amount of RAM). The vast majority of deployments will not benefit from heap sizes larger than 8GB.

Many users new to Cassandra are tempted to turn this value up immediately to consume the majority of the underlying system's RAM. Doing so in most cases is actually detrimental. The reason for this is that Cassandra, being essentially a database, spends a lot of time interacting with the operating system's I/O infrastructure (via the JVM of course). Modern operating systems maintain disk caches for frequently accessed data and are *very* good at keeping this data in memory. Regardless of how much RAM your hardware has, you should keep the JVM heap size constrained by the following formula and allow the operating system's file cache to do the rest:

(*memtable_total_space_in_mb*) + 1GB + (*key_cache_size_estimate*)

Note

As of Cassandra 1.0, column family row caches are stored in native memory by default (outside of the Java heap). This results in both a smaller per-row memory footprint and reduced JVM heap requirements, which helps keep the heap size manageable for good JVM garbage collection performance.

Tuning Java Garbage Collection

Cassandra's `GCInspector` class will log information about garbage collection whenever a garbage collection takes longer than 200ms. If garbage collections are occurring frequently and are taking a moderate length of time to complete (such as ConcurrentMarkSweep taking a few seconds), this is an indication that there is a lot of garbage collection pressure on the JVM; this needs to be addressed by adding nodes, lowering cache sizes, or adjusting the JVM options regarding garbage collection.

Tuning Compaction

During normal operations, numerous SSTables may be created on disk for a given column family. Compaction is the process of merging multiple SSTables into one consolidated SSTable. Additionally, the compaction process merges keys, combines columns, discards tombstones and creates a new index in the merged SSTable.

Choosing a Column Family Compaction Strategy

Tuning compaction involves first choosing the right compaction strategy for each column family based on its access patterns. As of Cassandra 1.0, there are two choices of compaction strategies:

- **SizeTieredCompactionStrategy** - This is the default compaction strategy for a column family, and prior to Cassandra 1.0, the only compaction strategy available. This strategy is best suited for column families with insert-mostly workloads that are not read as frequently. This strategy also requires closer monitoring of disk utilization because (as a worst case scenario) a column family can temporarily double in size while a compaction is in progress.
- **LeveledCompactionStrategy** - This is a new compaction strategy introduced in Cassandra 1.0. This compaction strategy is based on (but not an exact implementation of) Google's leveldb. This strategy is best suited for column families with read-heavy workloads that also have frequent updates to existing rows. When using this strategy, you want to keep an eye on read latency performance for the column family. If a node cannot keep up with the write workload and pending compactions are piling up, then read performance will degrade for a longer period of time.

Setting the Compaction Strategy on a Column Family

You can set the compaction strategy on a column family by setting the *compaction_strategy* attribute. For example, to update a column family to use the leveled compaction strategy using Cassandra CLI:

```
[default@demo] UPDATE COLUMN FAMILY users WITH compaction_strategy=LeveledCompactionStrategy
AND compaction_strategy_options=[{sstable_size_in_mb: 10}];
```

Tuning Options for Size-Tiered Compaction

For column families that use size-tiered compaction (the default), the frequency and scope of minor compactions is controlled by the following column family attributes:

- *min_compaction_threshold*
- *max_compaction_threshold*

These parameters set thresholds for the number of similar-sized SSTables that can accumulate before a minor compaction is triggered. With the default values, a minor compaction may begin any time after four SSTables are created on disk for a column family, and *must* begin before 32 SSTables accumulate.

You can tune these values per column family. For example, using Cassandra CLI:

```
[default@demo] UPDATE COLUMN FAMILY users WITH max_compaction_threshold = 20;
```

Note

Administrators can also initiate a major compaction through *nodetool compact*, which merges all SSTables into one. Though major compaction can free disk space used by accumulated SSTables, during runtime it temporarily doubles disk space usage and is I/O and CPU intensive. Also, once you run a major compaction, automatic minor compactions are no longer triggered frequently forcing you to manually run major compactions on a routine basis. So while read performance will be good immediately following a major compaction, it will continually degrade until the next major compaction is manually invoked. For this reason, major compaction is NOT recommended by DataStax.

Managing a Cassandra Cluster

This section discusses routine management and maintenance tasks.

Running Routine Node Repair

The *nodetool repair* command repairs inconsistencies across all of the replicas for a given range of data. Repair should be run at regular intervals during normal operations, as well as during node recovery scenarios (bringing a node back into the cluster after a failure).

Unless Cassandra applications perform no deletes at all, production clusters must schedule repair to run periodically on all nodes. The hard requirement for repair frequency is the value used for *gc_grace_seconds* -- make sure you run a repair operation at least once on each node within this time period. Following this important guideline can ensure that deletes are properly handled in the cluster.

Note

Repair is an expensive operation in both disk and CPU consumption. Use caution when running node repair on more than one node at a time, and schedule regular repair operations for low-usage hours.

In systems that seldom delete or overwrite data, it is possible to raise the value of *gc_grace_seconds* at a minimal cost in extra disk space used. This allows wider intervals for scheduling repair operations with the nodetool utility.

Adding Capacity to an Existing Cluster

Cassandra allows you to add capacity to a cluster by introducing new nodes to the cluster in stages. When a new node joins an existing cluster, it needs to know:

- Its position in the ring and the range of data it is responsible for. This is determined by the settings of *initial_token* and *auto_bootstrap* when the node first starts up.
- The nodes it should contact to learn about the cluster and establish the gossip process. This is determined by the setting of *seeds* when the node first starts up.

- The name of the cluster it is joining and how the node should be addressed within the cluster. See *Node and Cluster Initialization Properties* in *cassandra.yaml*.
- Any other non-default adjustments made to *cassandra.yaml* on your existing cluster should also be made on the new node as well before it is started.

Calculating Tokens For the New Nodes

When you add a node to a cluster, it needs to know its position in the ring. There are a few different approaches for calculating tokens for new nodes:

- **Add capacity by doubling the cluster size.** Adding capacity by doubling (or tripling or quadrupling) the number of nodes is operationally less complicated when assigning tokens. Existing nodes can keep their existing token assignments, and new nodes are assigned tokens that bisect (or trisect) the existing token ranges. For example, when you generate tokens for 6 nodes, three of the generated token values will be the same as if you generated for 3 nodes. You just need to determine the token values that are already in use, and assign the newly calculated token values to the newly added nodes.
- **Recalculate new tokens for all nodes and move nodes around the ring.** If doubling the cluster size is not feasible, and you need to increase capacity by a non-uniform number of nodes, you will have to recalculate tokens for the entire cluster. Existing nodes will have to have their new tokens assigned using *nodetool move*. After all nodes have been restarted with their new token assignments, run a *nodetool cleanup* in order to remove unused keys on all nodes. These operations are resource intensive and should be planned for low-usage times.
- **Add one node at a time and automatically assign a token with auto bootstrap. Use DataStax OpsCenter Enterprise Edition to rebalance your cluster.** When a node is started with *auto_bootstrap* set to `true` and *initial_token* left empty, Cassandra will split the token range of the heaviest loaded node and place the new node into the ring at that token position. Note that this approach will probably not result in a perfectly balanced ring, but it will alleviate hot spots. For DataStax Enterprise customers, you can quickly add nodes to the cluster using this approach and then use the rebalance feature of DataStax OpsCenter to automatically calculate balanced token ranges, move tokens accordingly, and then perform cleanup on the nodes after the moves are complete.

Adding Nodes to a Cluster

1. Install Cassandra on the new node, but do not start it.
2. Calculate tokens based on the expansion strategy you are using. If you want a new node to automatically pick a token range during auto bootstrap, you can skip this step.
3. In the *Node and Cluster Configuration (cassandra.yaml)* file, set *auto_bootstrap* to true. Set the other *Node and Cluster Initialization Properties* accordingly. Set *initial_token* according to your token calculations (or leave it unset to auto bootstrap the node into the ring by splitting the token range of the heaviest loaded node).
4. Start Cassandra on the new node. Allow a few minutes between node initializations. You can monitor the startup and data streaming process to its completion using *nodetool netstats*.
5. After the new nodes are fully bootstrapped, move the tokens on the existing nodes that require a new token assignment, one node at a time. First edit *initial_token* in the *Node and Cluster Configuration (cassandra.yaml)* file. Then run `nodetool move <new_token>` on the existing nodes that require a new token assignment, one node at a time.
6. After all nodes have their new tokens assigned, run *nodetool cleanup* on each of the existing nodes to remove the keys no longer belonging to those nodes. Wait for cleanup to complete on one node before doing the next. Cleanup may be safely postponed for low-usage hours.

Note

DataStax Enterprise customers can use DataStax OpsCenter Enterprise Edition to automatically rebalance their cluster after adding new nodes. OpsCenter's rebalance feature will automatically calculate balanced token ranges and perform steps 5 and 6 on each node in the cluster in the correct order.

Changing the Replication Factor

Increasing the replication factor increases the total number of copies of keyspace data stored in a Cassandra cluster.

1. Update each keyspace in the cluster and change its replication strategy options. For example, to update the number of replicas in Cassandra CLI when using SimpleStrategy replica placement strategy:

   ```
   [default@unknown] UPDATE KEYSPACE demo
   WITH strategy_options = [{replication_factor:3}];
   ```

   Or if using NetworkTopologyStrategy:

   ```
   [default@unknown] UPDATE KEYSPACE demo
   WITH strategy_options = [{datacenter1:6,datacenter2:6}];
   ```

2. On each node in the cluster, run *nodetool repair* for each keyspace that was updated. Wait until repair completes on a node before moving to the next node.

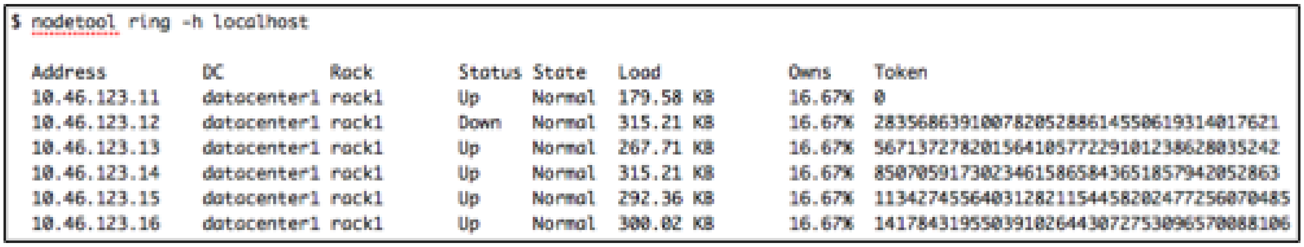
Replacing a Dead Node

To replace a node that has died (due to hardware failure, for example), you can bring up a new node in its place by starting the new node with the `-Dcassandra.replace_token=<token>` parameter and having the new node assume the token position of the node that has died. To replace a dead node in this way:

- the token that is used has to be from a node that is down - trying to replace a node using a token from a live node will result in an exception.
- the token that is used must already be part of the ring.
- the new node that is joining the cluster cannot have any preexisting Cassandra data on it (empty the data directory if you want to force a node replacement).

To replace a dead node:

1. Confirm the dead node using the *nodetool ring* command on any live node in the cluster (note the *Down* status and the token value of the dead node). For example:



```
$ nodetool ring -h localhost

  Address         DC          Rack        Status State   Load            Owns    Token
  10.46.123.11    datacenter1 rack1       Up     Normal  179.58 KB       16.67%  0
  10.46.123.12    datacenter1 rack1       Down   Normal  315.21 KB       16.67%  28356863910078205288614550619314017621
  10.46.123.13    datacenter1 rack1       Up     Normal  267.71 KB       16.67%  56713727820156410577229101238628035242
  10.46.123.14    datacenter1 rack1       Up     Normal  315.21 KB       16.67%  85070591730234615865843651857942052863
  10.46.123.15    datacenter1 rack1       Up     Normal  292.36 KB       16.67%  113427455640312821154458202477256070485
  10.46.123.16    datacenter1 rack1       Up     Normal  300.02 KB       16.67%  141784319550391026443072753096570088106
```

2. Prepare the replacement node by *installing Cassandra* and correctly configuring its *cassandra.yaml* file.
3. Start Cassandra on the new node using the startup property `-Dcassandra.replace_token=<token>` and pass in the same token that was used by the dead node. For example:

   ```
   $ cassandra -Dcassandra.replace_token=28356863910078205288614550619314017621
   ```

4. The new node will start in a hibernate state and begin to bootstrap data from its associated replica nodes. During this time, the node will not accept writes and is seen as down to other nodes in the cluster. When the bootstrap is complete, the node will be marked as up and any missed writes that occurred during bootstrap will be replayed using hinted handoff.
5. Once the new node is up, it is strongly recommended to run *nodetool repair* on each keyspace to ensure the node is fully consistent. For example:

```
$ nodetool repair -h 10.46.123.12 keyspace_name -pr
```

Backing Up and Restoring Data

Cassandra backs up data by taking a snapshot of the SSTable data files stored in the data directory. Snapshots are taken per keyspace, and can be taken while the system is online. However, nodes must be taken offline in order to restore a snapshot.

Using a parallel ssh tool (such as `pssh`), you can snapshot an entire cluster. This provides an *eventually consistent* backup. Although no one node is guaranteed to be consistent with its replica nodes at the time a snapshot is taken, a restored snapshot can resume consistency using Cassandra's built-in consistency mechanisms.

After a system-wide snapshot has been taken, you can enable incremental backups on each node (it is disabled by default) to backup data that has changed since the last snapshot was taken. Each time an SSTable is flushed, a hard link is copied into a `/backups` subdirectory of the data directory.

Taking a Snapshot

Snapshots are taken per node using the :ref:` nodetool snapshot <nodetool-snapshot>` command. If you want to take a global snapshot (capture all nodes in the cluster at the same time), run the `nodetool snapshot` command using a parallel ssh utility, such as `pssh`.

The snapshot command first flushes all in-memory writes to disk, then makes a copy of all on-disk data files (SSTable files) for each keyspace. The snapshot files are created in the Cassandra data directory location (`/var/lib/cassandra/data` by default) in the `snapshots` directory of each keyspace.

You must have enough disk space on the node in order to accommodate making a full copy of your data files. After the snapshot is complete, you can move the backup files off to another location if needed, or you can leave them in place.

To create a snapshot of a node

Run the `nodetool snapshot` command, specifying the hostname, JMX port and snapshot name. For example:

```
$ nodetool -h localhost -p 7199 snapshot 12022011
```

The snapshot will be created in `<data_directory_location>/<keyspace_name>/snapshots/<snapshot_name>`. Each snapshot folder contains numerous `.db` files which contain the data at the time of the snapshot.

Clearing Snapshot Files

When you take a snapshot, previous snapshot files are not automatically deleted. To maintain the snapshot directories, old snapshots that are no longer needed should be removed.

The *nodetool clearsnapshot* command will remove all existing snapshot files from the snapshot directory of each keyspace. You may want to make it part of your back-up process to clear old snapshots before taking a new one.

If you want to clear snapshots on all nodes at once, run the `nodetool clearsnapshot` command using a parallel ssh utility, such as `pssh`.

To clear all snapshots for a node

Run the `nodetool clearsnapshot` command. For example:

```
$ nodetool -h localhost -p 7199 clearsnapshot
```

Enabling Incremental Backups

When incremental backups are enabled (they are disabled by default), Cassandra hard-links each flushed SSTable to a backups directory under the keyspace data directory. This allows you to store backups offsite without transferring entire snapshots. Also, incremental backups combine with snapshots to provide a dependable, up-to-date backup mechanism.

To enable incremental backups, edit the `cassandra.yaml` file on each node in the cluster and change the value of *incremental_backups* to `true`.

As with snapshots, Cassandra does not automatically clear incremental backup files. DataStax recommends setting up a process to clear incremental backup hard-links each time a new snapshot is created.

Restoring from a Snapshot

To restore a keyspace from a snapshot, you will need all of the snapshot files for the keyspace, in addition to any incremental backup files created after the snapshot was taken (if using incremental backups).

If you are restoring just a single node, you must shutdown the node in order to restore. If you are restoring an entire cluster, you must shutdown all nodes, restore the snapshot data, and then start all nodes again.

Note

Restoring from snapshots and incremental backups will temporarily cause intensive CPU and I/O activity on the node being restored.

To restore a node from a snapshot and incremental backups:

1. Shut down the node to be restored.
2. Clear all files under the `commitlog` directory, (in `/var/lib/cassandra/commitlog` by default).
3. Clear all `*.db` files under `$DATA_DIRECTORY/<keyspace_name>`, but **DO NOT** delete the `/snapshots` and `/backups` subdirectories.
4. Locate the most recent snapshot folder under `$DATA_DIRECTORY/<keyspace_name>/snapshots/<snapshot_name>`, and copy its contents into `$DATA_DIRECTORY/<keyspace_name>`.
5. If using incremental backups as well, copy all contents of `$DATA_DIRECTORY/<keyspace_name>/backups` into `$DATA_DIRECTORY/<keyspace_name>`.
6. Restart the node, keeping in mind that a temporary burst of I/O activity will consume a large amount of CPU resources.

References

CQL Language Reference

Cassandra Query Language (CQL) is based on SQL (Structured Query Language), the standard for relational database manipulation. Although CQL has many similarities to SQL, there are some fundamental differences. For example, it is adapted to the Cassandra data model and architecture so there is still no allowance for SQL-like operations such as JOINs or range queries over rows on clusters that use the random partitioner.

CQL Lexical Structure

CQL input consists of a sequence of statements. A statement is composed of a sequence of tokens, terminated by a semicolon (`;`). Which tokens are valid depends on the syntax of the particular CQL command.

A token can be a keyword, an identifier, a constant (or literal) value, or a special character symbol. Tokens are normally separated by whitespace (space, tab, newline).

For example, the following is (syntactically) valid CQL input:

```
SELECT * FROM MyColumnFamily;

UPDATE MyColumnFamily SET 'SomeColumn' = 'SomeValue' WHERE KEY = B70DE1D0-9908-4AE3-BE34-5573E5B09F14;
```

This is a sequence of two CQL statements. This example shows one statement per line, although a statement can usefully be split across lines as well.

CQL Identifiers and Keywords

Tokens such as `SELECT`, `UPDATE`, or `FROM` are examples of keywords, that is, words that have a fixed meaning in a CQL statement. The token `MyColumnFamily` is an example of an identifier; a string that identifies the names of Cassandra objects such as keyspaces and column families.

CQL keywords are not case-sensitive, although they are typically shown in all uppercase in examples for readability. The keyword `SELECT` and `select` are equivalent.

An identifier must begin with a letter followed by any sequence of letters, digits, or the underscore (_). Identifiers are case-sensitive. The identifier `MyColumnFamily` and `mycolumnfamily` are not equivalent.

CQL Constants

A constant, also known as a literal or a scalar value, represents a specific data value. The format of a constant depends on the data type of the value it represents.

There are four types of *implicitly-typed* constants in CQL: string values, integer and float numeric values, and universally unique identifier (UUID) values. Constants can also be *explicitly* typed.

- **string** - A string constant is an arbitrary sequence of characters bounded by single quotes ('). To include a single-quote character within a string constant, use two adjacent single quotes (for example, 'Joe''s Diner'). Note that this is not the same as a double-quote character (").
- **integer** - An integer constant is one or more digits (0 through 9) preceded by an optional minus sign (-). Note that there is no optional plus sign (+). There is also no provision for exponential notation such as 5e2.
- **float** - A float constant is specified as `<digits>.<digits>`, where `<digits>` is one or more digits ((0 through 9). Note that there must be digits both before and after the decimal point (for example 0.12 instead of .12). There is also no provision for exponential notation such as 1.925e-3.
- **uuid** - A universally unique identifier can be expressed in the canonical UUID form: 32 hex digits (0-9 or a-f, case insensitive), separated by dashes (-) after the 8th, 12th, 16th, and 20th digits. For example: 01234567-0123-0123-0123-0123456789ab

CQL Comments

Comments can be used to document CQL statements in your application code. Single line comments can begin with a double dash (`--`) or a double slash (`//`) and extend to the end of the line. Multi-line comments can be enclosed in `/*` and `*/` characters.

CQL Consistency Levels

In Cassandra, consistency refers to how up-to-date and synchronized a row of data is on all of its replica nodes. For any given read or write operation, the client request specifies a consistency level, which determines how many replica nodes must successfully respond to the request.

In CQL, the default consistency level is `ONE`. You can set the consistency level for any read (SELECT) or write (INSERT, UPDATE, DELETE, BATCH) operation using the `USING CONSISTENCY` keywords. For example:

```
SELECT * FROM users WHERE state='TX' USING CONSISTENCY QUORUM;
```

The following consistency levels can be specified. See *tunable consistency* for more information about the different consistency levels.

- ANY (applicable to writes only)
- ONE
- QUORUM
- LOCAL_QUOROM (applicable to multi-data center clusters only)
- EACH_QUOROM (applicable to multi-data center clusters only)

CQL Data Types

Cassandra has a schema-optional data model. You can define data types when you create your column family schemas (which is recommended), but Cassandra does not require it. Column names, column values, and row key values can be typed in Cassandra. Internally, Cassandra stores column names and values as hex byte arrays (`blob`).

CQL comes with the following built-in data types, which can be used for column names and column/row key values. One exception is `counter`, which is only allowed as a column value (not allowed for row key values or column names).

CQL Type	Description
ascii	US-ASCII character string
bigint	64-bit signed long
blob	Arbitrary hexadecimal bytes (no validation)
boolean	true or false
counter	Distributed counter value (64-bit long)
decimal	Variable-precision decimal
double	64-bit IEEE-754 floating point
float	32-bit IEEE-754 floating point
int	32-bit signed integer
text	UTF-8 encoded string
timestamp	Date plus time, encoded as 8 bytes since epoch
uuid	Type 1 or type 4 UUID
varchar	UTF-8 encoded string
varint	Arbitrary-precision integer

Working with Dates and Times

Values serialized with the `timestamp` type are encoded as 64-bit signed integers representing a number of milliseconds since the standard base time known as the *epoch*: January 1 1970 at 00:00:00 GMT.

Timestamp types can be input in CQL as simple long integers, giving the number of milliseconds since the epoch.

Timestamp types can also be input as string literals in any of the following ISO 8601 formats. For example, for the date and time of Jan 2, 2003, at 04:05:00 AM, GMT:

```
2011-02-03 04:05+0000
2011-02-03 04:05:00+0000
2011-02-03T04:05+0000
2011-02-03T04:05:00+0000
```

The `+0000` is the RFC 822 4-digit time zone specification for GMT. US Pacific Standard Time is `-0800`. The time zone may be omitted. For example:

```
2011-02-03 04:05
2011-02-03 04:05:00
2011-02-03T04:05
2011-02-03T04:05:00
```

If no time zone is specified, the time zone of the Cassandra coordinator node handing the write request will be used. For accuracy, DataStax recommends specifying the time zone rather than relying on the time zone configured on the Cassandra nodes.

If you only want to capture date values, the time of day can also be omitted. For example:

```
2011-02-03
2011-02-03+0000
```

In this case, the time of day will default to 00:00:00, in the specified or default time zone.

CQL Storage Parameters

Certain CQL commands allow a `WITH` clause for setting certain storage parameters on a keyspace or column family. Note that CQL does not currently offer support for defining *all of the possible storage parameters*, just a subset.

CQL Keyspace Storage Parameters

CQL supports setting the following keyspace storage parameters.

CQL Parameter Name	Description
strategy_class	The name of the replication strategy: `SimpleStrategy` or `NetworkTopologyStrategy`
strategy_options	Replication strategy option names are appended to the `strategy_options` keyword using a colon (:). For example: `strategy_options:DC1=1` or `strategy_options:replication_factor=3`

CQL Column Family Storage Parameters

CQL supports setting the following column family storage parameters.

CQL Parameter Name	CQLSH Default Value
comparator	text
comment	n/a
row_cache_provider	`SerializingCacheProvider` if JNA is present, otherwise `ConcurrentHashMapCacheProvider`
row_cache_size	0
key_cache_size	200000
read_repair_chance	1.0
gc_grace_seconds	864000
default_validation	text
min_compaction_threshold	4
max_compaction_threshold	32
row_cache_save_period_in_seconds	0
key_cache_save_period_in_seconds	14400
replicate_on_write	false

CQL Commands

The CQL language is comprised on the following commands:

ALTER COLUMNFAMILY

Manipulates the column metadata of a column family.

Synopsis

```
ALTER COLUMNFAMILY <name>
   ALTER <column_name> TYPE <data_type>
  | ADD <column_name> <data_type>
  | DROP <column_name> ;
```

Description

`ALTER COLUMNFAMILY` is used to manipulate the column metadata of a static column family. You can also use the alias `ALTER TABLE`. Using the alias, you can add new columns, drop existing columns, or change the data storage type of existing columns. No results are returned.

See *CQL Data Types* for the available data types.

Parameters

<name>

The name of the column family to be altered.

ALTER <column_name> TYPE <data_type>

Changes the data type of an existing column. The named column must exist in the column family schema definition and have a type defined, but the column does not have to exist in any rows currently stored in the column family. Note that when you change the data type of a column, existing data is not changed or validated on disk. If existing data is not compatible with the defined type, then this will cause your CQL driver or other client interfaces to return errors when accessing the data. For example if you changed a column type to `int`, but the existing stored data had non-numeric characters in it, your client requests of that data would report errors.

ADD <column_name> <data_type>

Adds a typed column to the column metadata of a column family schema definition. The column must not already have a type in the column family metadata.

DROP <column_name> <data_type>

Removes a column from the column family metadata. Note that this does not remove the column from current rows. It just removes the metadata saying that the values stored under that column name are expected to be a certain type.

Examples

```
ALTER COLUMNFAMILY users ALTER email TYPE varchar;

ALTER COLUMNFAMILY users ADD gender varchar;

ALTER COLUMNFAMILY users DROP gender;
```

BATCH

Sets a global consistency level, client-supplied timestamp, and optional time-to-live (TTL) for all columns written by the statements in the batch.

Synopsis

```
BEGIN BATCH

   [ USING <write_option> [ AND <write_option> [...] ] ];
```

```
    <dml_statement>
    <dml_statement>
    [...]

APPLY BATCH;
```

where `<write_option>` is:

```
USING CONSISTENCY <consistency_level>
TTL <seconds>
TIMESTAMP <integer>
```

Description

A `BATCH` statement allows you combine multiple data modification statements into a single logical operation. All columns modified by the batch statement will have the same global timestamp. Only `INSERT`, `UPDATE`, and `DELETE` statements are allowed within a `BATCH` statement. Individual statements within a BATCH should be specified one statement per line without an ending semi-colon.

All statements within the batch are executed using the same consistency level.

BATCH should not be considered as an analogue for SQL ACID transactions. BATCH does not provide transaction isolation. Column updates are only considered atomic within a given record (row).

Parameters

BEGIN BATCH

Command to initiate a batch. All statements in the batch will be executed with the same timestamp and consistency level (and optional time-to-live).

USING CONSISTENCY <consistency_level>

Optional clause to specify the consistency level. If omitted, the default consistency level is ONE. The following consistency levels can be specified. See *tunable consistency* for more information about the different consistency levels.

- ANY (applicable to writes only)
- ONE
- QUORUM
- LOCAL_QUOROM (applicable to multi-data center clusters only)
- EACH_QUOROM (applicable to multi-data center clusters only)

TTL <seconds>

Optional clause to specify a time-to-live (TTL) period for an inserted or updated column. TTL columns are automatically marked as deleted (with a tombstone) after the requested amount of time has expired.

TIMESTAMP <integer>

Defines an optional timestamp to use for the written columns. The timestamp must be in the form of an integer.

<dml_statement>

An *INSERT*, *UPDATE*, or *DELETE* statement to be executed. A statement should be contained on a single line and without an ending semi-colon.

APPLY BATCH

Closes the batch statement. All statements in the batch are submitted for execution as a unit.

Example

```
BEGIN BATCH USING CONSISTENCY QUORUM AND TTL 8640000
  INSERT INTO users (KEY, password, name) VALUES ('user2', 'ch@ngem3b', 'second user')
  UPDATE users SET password = 'ps22dhds' WHERE KEY = 'user2'
  INSERT INTO users (KEY, password) VALUES ('user3', 'ch@ngem3c')
  DELETE name FROM users WHERE key = 'user2'
  INSERT INTO users (KEY, password, name) VALUES ('user4', 'ch@ngem3c', 'Andrew')
APPLY BATCH;
```

CREATE COLUMNFAMILY

Define a new column family.

Synopsis

```
CREATE COLUMNFAMILY <cf_name> (
  <key_column_name> <data_type> PRIMARY KEY
  [, <column_name> <data_type> [, ...] ] )
  [ WITH <storage_parameter> = <value>
  [AND <storage_parameter> = <value> [...] ] ];
```

Description

`CREATE COLUMNFAMILY` creates a new column family under the current keyspace. You can also use the alias `CREATE TABLE`.

The only schema information that must be defined for a column family is the primary key (or row key) and its associated data type. Other column metadata can be defined as needed.

See *CQL Data Types* for the available data types.

See *CQL Column Family Storage Parameters* for the available storage parameters you can define on a column family when using CQL.

Parameters

<cf_name>

Defines the name of the column family. Valid column family names are strings of alpha-numeric characters and underscores, and must begin with a letter.

<key_column_name> <data_type> PRIMARY KEY

Columns are defined in a comma-separated list enclosed in parenthesis. The first column listed in the column definition is always the row key (or primary key of the column family), and is required. Any other column definitions are optional.

Row keys can be defined using the generic `KEY` keyword, or can be given a column name to use as the alias for the row key. The row key data type must be compatible with the partitioner configured for your Cassandra cluster. For example, `OrderPreservingPartitioner` requires UTF-8 row keys.

<column_name> <data_type>

Defines column metadata for static column families (when you know what the column names will be ahead of time).

WITH <storage_parameter> = <value>

Defines certain storage parameters on a column family. See *CQL Column Family Storage Parameters* for the available storage parameters you can define on a column family when using CQL.

For dynamic column families (where you do not know the column names ahead of time), it is best practice to still define a default data type for column names (using `WITH comparator=<data_type>`) and values (using `WITH default_validation=<data_type>`).

Examples

Dynamic column family definition:

```
CREATE COLUMNFAMILY user_events (user text PRIMARY KEY)
   WITH comparator=timestamp AND default_validation=int;
```

Static column family definition:

```
CREATE COLUMNFAMILY users (
   KEY uuid PRIMARY KEY,
   username text,
   email text )
   WITH comment='user information'
   AND read_repair_chance = 1.0;
```

CREATE INDEX

Define a new secondary index on a single, typed column of a column family.

Synopsis

```
CREATE INDEX [<index_name>]
    ON <cf_name> (<column_name>);
```

Description

`CREATE INDEX` creates a secondary index on the named column family, for the named column. A secondary index can only be created on a single, typed column. The indexed column must have a data type defined in the column metadata of the column family definition, although it is not required that the column exist in any currently stored rows.

Parameters

<index_name>

Optionally defines a name for the secondary index. Valid index names are strings of alpha-numeric characters and underscores, and must begin with a letter.

ON <cf_name> (<column_name>)

Specifies the column family and column name on which to create the secondary index. The named column must have a data type defined in the column family schema definition.

Examples

Define a static column family and then create a secondary index on two of its named columns:

```
CREATE COLUMNFAMILY users (
   KEY uuid PRIMARY KEY,
   firstname text,
   lastname text,
   email text,
   address text,
   zip int,
   state text);

CREATE INDEX user_state
   ON users (state);
```

```
CREATE INDEX ON users (zip);
```

CREATE KEYSPACE

Define a new keyspace and its replica placement strategy.

Synopsis

```
CREATE KEYSPACE <ks_name>
    WITH strategy_class = <value>
    [ AND strategy_options:<option> = <value> [...] ];
```

Description

`CREATE KEYSPACE` creates a new keyspace and sets the replica placement strategy (and associated replication options) for the keyspace.

See *Choosing Keyspace Replication Options* for guidance on how to best configure replication strategy and strategy options for your cluster.

Parameters

<ks_name>

Defines the name of the keyspace. Valid keyspace names are strings of alpha-numeric characters and underscores, and must begin with a letter.

WITH strategy_class=<value>

Required. Sets the *replica placement strategy* to use for this keyspace. The most common choices are `NetworkTopologyStrategy` or `SimpleStrategy`.

AND strategy_options:<option>=<value>

Certain additional options must be defined depending on the replication strategy chosen.

For `SimpleStrategy`, you must specify the replication factor in the format of `strategy_options:replication_factor=<number>`.

For `NetworkTopologyStrategy`, you must specify the number of replicas per data center in the format of `strategy_options:<datacenter_name>=<number>`. Note that what you specify for <datacenter_name> depends on the cluster-configured *snitch* you are using. There is a correlation between the data center name defined in the keyspace `strategy_options` and the data center name as recognized by the snitch you are using. The *nodetool ring* command prints out data center names and rack locations of your nodes if you are not sure what they are.

Examples

Define a new keyspace using the simple replication strategy:

```
CREATE KEYSPACE MyKeyspace WITH strategy_class = 'SimpleStrategy'
 AND strategy_options:replication_factor = 1;
```

Define a new keyspace using a network-aware replication strategy and snitch. This example assumes you are using the *PropertyFileSnitch* and your data centers are named `DC1` and `DC2` in the `cassandra-topology.properties` file:

```
CREATE KEYSPACE MyKeyspace WITH strategy_class = 'NetworkTopologyStrategy'
 AND strategy_options:DC1 = 3 AND strategy_options:DC2 = 3;
```

DELETE

Removes one or more columns from the named row(s).

Synopsis

```
DELETE [<column_name> [, ...]]
  FROM <column_family>
[USING CONSISTENCY <consistency_level> [AND TIMESTAMP <integer>]]
WHERE <row_specification>;
```

where `<row_specification>` is:

```
KEY | <key_alias> = <key_value>
KEY | <key_alias> IN (<key_value> [,...])
```

Description

A `DELETE` statement removes one or more columns from one or more rows in the named column family. Rows are identified using the `KEY` keyword or the key alias defined on the column family. If no column names are given, the entire row is deleted.

When a column is deleted, it is not removed from disk immediately. The deleted column is marked with a tombstone and then removed after the configured grace period has expired. See *About Deletes* for more information about how Cassandra handles deleted columns and rows.

Parameters

<column_name>

The name of one or more columns to be deleted. If no column names are given, then all columns in the identified rows will be deleted, essentially deleting the entire row.

FROM <column_family>

The name of the column family from which to delete the identified columns or rows.

USING CONSISTENCY <consistency_level>

Optional clause to specify the consistency level. If omitted, the default consistency level is ONE. The following consistency levels can be specified. See *tunable consistency* for more information about the different consistency levels.

- ANY (applicable to writes only)
- ONE
- QUORUM
- LOCAL_QUOROM (applicable to multi-data center clusters only)
- EACH_QUOROM (applicable to multi-data center clusters only)

TIMESTAMP <integer>

Defines an optional timestamp to use for the new tombstone record. The timestamp must be in the form of an integer.

WHERE <row_specification>

The `WHERE` clause identifies one or more rows in the column family from which to delete columns. Rows are identified using the `KEY` keyword or the key alias defined on the column family, and then supplying one or more row key values.

Example

```
DELETE email, phone
  FROM users
  USING CONSISTENCY QUORUM AND TIMESTAMP 1318452291034
  WHERE user_name = 'jsmith';

DELETE FROM users WHERE KEY IN ('dhutchinson', 'jsmith');
```

DROP COLUMNFAMILY

Drops the named column family.

Synopsis

```
DROP COLUMNFAMILY <name>;
```

Description

A `DROP COLUMNFAMILY` statement results in the immediate, irreversible removal of a column family, including all data contained in the column family. You can also use the alias `DROP TABLE`.

Parameters

<name>

The name of the column family to be dropped.

Example

```
DROP COLUMNFAMILY users;
```

DROP INDEX

Drops the named secondary.

Synopsis

```
DROP INDEX <name>;
```

Description

A `DROP INDEX` statement removes an existing secondary index.

Parameters

<name>

The name of the secondary index to be dropped. If the index was not given a name during creation, the index name is `<columnfamily_name>_<column_name>_idx`.

Example

```
DROP INDEX user_state;

DROP INDEX users_zip_idx;
```

DROP KEYSPACE

Drops the named keyspace.

Synopsis

```
DROP KEYSPACE <name>;
```

Description

A `DROP KEYSPACE` statement results in the immediate, irreversible removal of a keyspace, including all column families and data contained in the keyspace.

Parameters

<name>

The name of the keyspace to be dropped.

Example

```
DROP KEYSPACE Demo;
```

INSERT

Inserts or updates one or more columns in the identified row of a column family.

Synopsis

```
INSERT INTO <column_family> (<key_name>, <column_name> [, ...])
 VALUES (<key_value>, <column_value> [, ...])
 [USING <write_option> [AND <write_option> [...] ] ];
```

where `<write_option>` is:

```
USING CONSISTENCY <consistency_level>
TTL <seconds>
TIMESTAMP <integer>
```

Description

An `INSERT` is used to write one or more columns to the identified row in a Cassandra column family. No results are returned. Unlike in SQL, the semantics of `INSERT` and `UPDATE` are identical. In either case a row/column record is created if does not exist, or updated if it does exist.

The first column name in the INSERT list must be that of the row key (either the `KEY` keyword or the row key alias defined on the column family). The first column value in the VALUES list is the row key value for which you want to insert or update columns. After the row key, there must be at least one other column name specified. In Cassandra, a row with only a key and no associated columns is not considered to exist.

Parameters

<column_family> (<key_name>, <column_name> [, ...])

The name of the column family to insert or update followed by a comma-separated list of column names enclosed in parenthesis. The first name in the list must be that of the row key (either the `KEY` keyword or the row key alias defined on the column family) followed by at least one other column name.

VALUES (<key_value>, <column_value> [, ...])

Supplies a comma-separated list of column values enclosed in parenthesis. The first column value is always the row key value for which you want to insert columns. Column values should be listed in the same order as the column names supplied in the INSERT list. If a row or column does not exist, it will be inserted. If it does exist, it will be updated.

USING CONSISTENCY <consistency_level>

Optional clause to specify the consistency level. If omitted, the default consistency level is ONE. The following consistency levels can be specified. See *tunable consistency* for more information about the different consistency levels.

- ANY (applicable to writes only)
- ONE
- QUORUM
- LOCAL_QUOROM (applicable to multi-data center clusters only)
- EACH_QUOROM (applicable to multi-data center clusters only)

TTL <seconds>

Optional clause to specify a time-to-live (TTL) period for an inserted or updated column. TTL columns are automatically marked as deleted (with a tombstone) after the requested amount of time has expired.

TIMESTAMP <integer>

Defines an optional timestamp to use for the written columns. The timestamp must be in the form of an integer.

Example

```
INSERT INTO users (KEY, user_name)
   VALUES ('cfd66ccc-d857-4e90-b1e5-df98a3d40cd6', 'jbellis')
   USING CONSISTENCY LOCAL_QUORUM AND TTL 86400;
```

SELECT

Returns the requested rows and columns from a Cassandra column family.

Synopsis

```
SELECT <column_specification>
   FROM [<keyspace>.]<column_family>
[USING CONSISTENCY <consistency_level>]
[WHERE <row_specification> [AND <row_specification> [...]]
[LIMIT <integer>]
```

where `<column_specification>` is:

```
<column_name> [, ...]
| [FIRST <integer>] [REVERSED] { <start_of_range> .. <end_of_range> | * }
| COUNT(*)
```

and where `<row_specification>` is:

```
KEY | <key_alias> { = | < | > | <= | >= } <key_value>
KEY | <key_alias> IN (<key_value> [,...])
```

Description

A `SELECT` is used to read one or more rows from a Cassandra column family. It returns a result-set of rows, where each row consists of a row key and a collection of columns corresponding to the query.

Unlike a SQL SELECT, there is no guarantee that the columns specified in the query will be contained in the result set. Cassandra has a schema-optional data model, so it will not give an error if you request columns that do not exist.

Parameters

The SELECT List

The SELECT list determines the columns that will appear in the results. It takes a comma-separated list of column names, a range of column names, or `COUNT(*)`. Column names in Cassandra can be specified as string literals or integers, in addition to named identifiers.

To specify a range of columns, specify the start and end column names separated by two periods (..). The set of columns returned for a range is start and end inclusive. The asterisk (*) may also be used as a range representing all columns.

When requesting a range of columns, it may be useful to limit the number of columns that can be returned from each row using the `FIRST` clause. This sets an upper limit on the number of columns returned per row (the default is 10,000 if not specified).

The `REVERSED` keyword causes the columns to be returned in reversed sorted order. If using a `FIRST` clause, the columns at the end of the range will be selected instead of the ones at the beginning of the range.

A SELECT list may also be `COUNT(*)`. In this case, the result will be the number of rows which matched the query.

The FROM Clause

The `FROM` clause specifies the column family to query. If a keyspace is not specified, the current keyspace will be used.

USING CONSISTENCY <consistency_level>

Optional clause to specify the consistency level. If omitted, the default consistency level is ONE. The following consistency levels can be specified. See *tunable consistency* for more information about the different consistency levels.

- ONE
- QUORUM
- LOCAL_QUOROM (applicable to multi-data center clusters only)
- EACH_QUOROM (applicable to multi-data center clusters only)

The WHERE Clause

The `WHERE` clause filters the rows that appear in the results. You can filter on a key name, a range of keys, or on column values (in the case of columns that have a secondary index). Row keys are specified using the `KEY` keyword or key alias defined on the column family, followed by a relational operator (one of =, >, >=, <, or <=), and then a value. When terms appear on both sides of a relational operator it is assumed the filter applies to an indexed column. With column index filters, the term on the left of the operator must be the name of the indexed column, and the term on the right is the value to filter on.

Note: The greater-than and less-than operators (> and <) result in key ranges that are inclusive of the terms. There is no supported notion of strictly greater-than or less-than; these operators are merely supported as aliases to >= and <=.

The LIMIT Clause

The LIMIT clause limits the number of rows returned by the query. The default is 10,000 rows.

Examples

Select two columns from three rows:

```
SELECT name, title FROM employees WHERE KEY IN (199, 200, 207);
```

Select a range of columns from all rows, but limit the number of columns to 3 per row starting with the end of the range:

```
SELECT FIRST 3 REVERSED 'time199'..'time100' FROM events;
```

Count the number of rows in a column family:

```
SELECT COUNT(*) FROM users;
```

TRUNCATE

Removes all data from a column family.

Synopsis

```
TRUNCATE <column_family>;
```

Description

A `TRUNCATE` statement results in the immediate, irreversible removal of all data in the named column family.

Parameters

<column_family>

The name of the column family to truncate.

Example

```
TRUNCATE user_activity;
```

UPDATE

Updates one or more columns in the identified row of a column family.

Synopsis

```
UPDATE <column_family>
   [ USING <write_option> [ AND <write_option> [...] ] ];
     SET <column_name> = <column_value> [, ...]
       | <counter_column_name> = <counter_column_name> {+ | -} <integer>
     WHERE <row_specification>;
```

where `<write_option>` is:

```
USING CONSISTENCY <consistency_level>
TTL <seconds>
TIMESTAMP <integer>
```

and where `<row_specification>` is:

```
KEY | <key_alias> = <key_value>
KEY | <key_alias> IN (<key_value> [,...])
```

Description

An `UPDATE` is used to update or write one or more columns to the identified row in a Cassandra column family. Row/column records are created if they do not exist, or updated if they do exist.

Rows are created or updated by supplying column names and values, after the `SET` keyword. Multiple columns can be set by separating the name/value pairs using commas. Each update statement requires a precise set of row keys to be specified using a `WHERE` clause. Rows are identified using the `KEY` keyword or the key alias defined on the column family.

Parameters

<column_family>

The name of the column family from which to update (or insert) the identified columns and rows.

USING CONSISTENCY <consistency_level>

Optional clause to specify the consistency level. If omitted, the default consistency level is ONE. The following consistency levels can be specified. See *tunable consistency* for more information about the different consistency levels.

- ANY (applicable to writes only)
- ONE
- QUORUM
- LOCAL_QUOROM (applicable to multi-data center clusters only)
- EACH_QUOROM (applicable to multi-data center clusters only)

TTL <seconds>

Optional clause to specify a time-to-live (TTL) period for an inserted or updated column. TTL columns are automatically marked as deleted (with a tombstone) after the requested amount of time has expired.

TIMESTAMP <integer>

Defines an optional timestamp to use for the written columns. The timestamp must be in the form of an integer.

SET

The `SET` clause is used to specify a comma-separated list of column name/value pairs that you want to update or insert. If the named column exists, its value will be updated. If it does not exist it will be inserted. For counter column families, you can update a counter column value by specifying an increment or decrement value to be applied to the current value of the counter column.

WHERE <row_specification>

The `WHERE` clause identifies one or more rows in the column family for which to update the named columns. Rows are identified using the `KEY` keyword or the key alias defined on the column family, and then supplying one or more row key values.

Example

Update a column in several rows at once:

```
UPDATE users USING CONSISTENCY QUORUM
    SET 'state' = 'TX'
    WHERE KEY IN (88b8fd18-b1ed-4e96-bf79-4280797cba80,
                  06a8913c-c0d6-477c-937d-6c1b69a95d43,
                  bc108776-7cb5-477f-917d-869c12dfffa8);
```

Update several columns in a single row:

```
UPDATE users USING CONSISTENCY QUORUM
    SET 'name' = 'John Smith', 'email' = 'jsmith@cassie.com'
    WHERE user_uuid = 88b8fd18-b1ed-4e96-bf79-4280797cba80;
```

Update the value of a counter column:

```
UPDATE page_views USING CONSISTENCY QUORUM AND TIMESTAMP=1318452291034
 SET 'index.html' = 'index.html' + 1
 WHERE KEY = 'www.datastax.com';
```

USE

Connects the current client session to a keyspace.

Synopsis

```
USE <keyspace_name>;
```

Description

A `USE` statement tells the current client session and the connected Cassandra instance what keyspace you will be working in. All subsequent operations on column families and indexes will be in the context of the named keyspace, unless otherwise specified or until the client connection is terminated or another `USE` statement is issued.

Parameters

<keyspace_name>

The name of the keyspace to connect to for the current client session.

Example

```
USE PortfolioDemo;
```

CQLSH-Specific Commands

CQL Shell (`cqlsh`) has these additional CQL commands available in the command-line interface:

ASSUME

Sets the client-side encoding for a `cqlsh` session.

Synopsis

```
ASSUME [<keyspace_name>].<columnfamily_name>
       <storage_type_definition>
       [, ...] ;

Where <storage_type_definition> is:

   .. code-block:: sql

       (KEY | <column_name>) VALUES ARE <datatype>
     | NAMES ARE <datatype>
     | VALUES ARE <datatype>
```

Description

Cassandra is a schema-optional data model, meaning that column families are not required to have data type information explicitly defined for column names, column values or row key values. When type information is not explicitly defined, and implicit typing cannot be determined, data is displayed as raw hex bytes (`blob` type), which is not human-readable. The ASSUME command allows you to specify type information for particular column family values passed between the `cqlsh` client and the Cassandra server.

Parameters

[<keyspace_name>].<columnfamily_name>

The name of the column family (and optionally the keyspace) for which you are specifying assumed types. If keyspace is not supplied, the keyspace is assumed to be the keyspace you are currently connected to.

NAMES ARE <datatype>

Used to declare an assumed type for column names.

VALUES ARE <datatype>

Used to declare an assumed type for column values.

(KEY | <column_name>) VALUES ARE <datatype>

Used to declare an assumed type for a particular column, such as the row key. If your column family row key does not have a name or alias defined, you can use the `KEY` keyword to denote the row key. If the row key (or any other column) has a name or alias defined in the column family schema, you can declare it by its name.

Examples

```
ASSUME users NAMES ARE text, VALUES are text;
```

```
ASSUME users(KEY) VALUES are uuid;
```

```
ASSUME users(user_id) VALUES are uuid;
```

DESCRIBE

Outputs information about the connected Cassandra cluster, or about the data objects stored in the cluster.

Synopsis

```
DESCRIBE CLUSTER
        | SCHEMA
        | KEYSPACE [<keyspace_name>]
        | COLUMNFAMILY <columnfamily_name>
```

Description

A `DESCRIBE` statement outputs information about the currently connected Cassandra cluster and the data objects (keyspaces and column families) stored in the cluster. It outputs CQL statements that can be used to recreate the database schema if needed.

Parameters

CLUSTER

Outputs information about the Cassandra cluster, such as the cluster name, partitioner, and snitch configured for the cluster. When connected to a non-system keyspace, also shows the data endpoint ranges owned by each node in the Cassandra ring.

SCHEMA

Outputs CQL commands that can be used to recreate the entire schema (all keyspaces and column families managed by the cluster).

KEYSPACE [<keyspace_name>]

Outputs CQL commands that can be used to recreate the keyspace and its column families. May also show metadata about the keyspace. If a keyspace name is not provided, the current keyspace is described.

COLUMNFAMILY <columnfamily_name>

Output CQL commands that can be used to recreate the column family schema. May also show metadata about the column family.

Examples

```
DESCRIBE CLUSTER;

DESCRIBE KEYSPACE PortfolioDemo;

DESCRIBE COLUMNFAMILY Stocks;
```

SHOW

Shows the Cassandra version, host, or data type assumptions for the current `cqlsh` client session.

Synopsis

```
SHOW VERSION
    | HOST
    | ASSUMPTIONS;
```

Description

A `SHOW` statement displays information about the current `cqlsh` client session, including the Cassandra version, the Cassandra host, and any data type assumptions that have been set.

Parameters

VERSION

Shows the version and build number of the connected Cassandra instance, as well as the versions of the CQL specification and the Thrift protocol that the connected Cassandra instance understands.

HOST

Shows the host information of the Cassandra node that the `cqlsh` session is currently connected to.

ASSUMPTIONS

Shows the data type assumptions for the current `cqlsh` session as specified by the `ASSUME` command.

Examples

```
SHOW VERSION;

SHOW HOST;

SHOW ASSUMPTIONS;
```

nodetool

Under the bin directory you will find the nodetool utility. This can be used to help manage a cluster. Usage:

```
bin/nodetool -h HOSTNAME [-p JMX_PORT ] COMMAND...
```

If a username and password for RMI authentication are set explicitly in the `cassandra-env.sh` file for the host, then you must specify credentials:

```
bin/nodetool -h HOSTNAME [-p JMX_PORT -u JMX_USERNAME -p JMX_PASSWORD ] COMMAND...
```

The available commands are:

ring

Displays node status and information about the ring as determined by the node being queried. This can give you a quick idea of how balanced the load is around the ring and if any nodes are down. If your cluster is not properly configured, different nodes may show a different ring, so this is a good way to check that every node views the ring the same way.

join

Causes the node to join the ring. This assumes that the node was initially started *not* in the ring, or in other words, started with `-Djoin_ring=false`. Note that the joining node should be properly configured with the desired options for seed list, initial token, and autoboostrapping.

info

Outputs node information including the token, load info (on disk storage), generation number (times started), uptime in seconds, and heap memory usage.

cfstats

Prints statistics for every keyspace and column family.

version

Prints the Cassandra release version for the node being queried.

cleanup [*keyspace*][*cf_name*]

Triggers the immediate cleanup of keys no longer belonging to this node. This has roughly the same effect on a node that a major compaction does in terms of a temporary increase in disk space usage and an increase in disk I/O. Optionally takes a list of column family names.

compact [*keyspace*][*cf_name*]

For column families that use the SizeTieredCompactionStrategy, initiates an immediate major compaction of all column families in *keyspace*. For each column family in *keyspace*, this compacts all existing SSTables into a single SSTable. This will cause considerable disk I/O and will temporarily cause up to twice as much disk space to be used. Optionally takes a list of column family names.

upgradesstables [*keyspace*][*cf_name*]

Rebuilds SSTables on a node for the named column families. Use when upgrading your server or changing compression options (available from Cassandra 1.0.4 onwards).

scrub [*keyspace*][*cf_name*]

Rebuilds SSTables on a node for the named column families and snapshots data files before rebuilding as a safety measure. If possible use `upgradesstables`. While `scrub` rebuilds SSTables, it also discards data that it deems broken and creates a snapshot, which you have to remove manually.

cfhistograms *keyspace cf_name*

Prints statistics on the read/write latency for a column family. These statistics, which include row size, column count and bucket offsets, can be useful for monitoring activity in a column family.

snapshot [*snapshot-name*]

Takes an online snapshot of Cassandra's data. Before taking the snapshot, the node is flushed. The results can be found in Cassandra's data directory (typically `/var/lib/cassandra/data`) under the `snapshots` directory of each keyspace. See also:

- http://wiki.apache.org/cassandra/Operations#Backing_up_data

clearsnapshot

Deletes all existing snapshots.

tpstats

Prints the number of active, pending, and completed tasks for each of the thread pools that Cassandra uses for stages of operations. A high number of pending tasks for any pool can indicate performance problems. For more details, see:

- http://wiki.apache.org/cassandra/Operations#Monitoring

flush *keyspace* [*cf_name*]

Flushes all memtables for a keyspace to disk, allowing the commit log to be cleared. Optionally takes a list of column family names.

drain

Flushes all memtables for a node and causes the node to stop accepting write operations. Read operations will continue to work. This is typically used before upgrading a node to a new version of Cassandra.

repair *keyspace* [*cf_name*] [-pr]

Begins an anti-entropy node repair operation. If the `-pr` option is specified, only the first range returned by the partitioner for a node will be repaired. This allows you to repair each node in the cluster in succession without duplicating work.

Without `-pr`, all replica ranges that the node is responsible for will be repaired.

Optionally takes a list of column family names.

decommission

Tells a live node to decommission itself (streaming its data to the next node on the ring) See also:

- http://wiki.apache.org/cassandra/NodeProbe#Decommission
- http://wiki.apache.org/cassandra/Operations#Removing_nodes_entirely

move *new_token*

Moves a node to a new token. This essentially combines decommission and bootstrap. See:

- http://wiki.apache.org/cassandra/Operations#Moving_nodes

loadbalance

Moves the node to a new token so that it will split the range of whatever token currently has the highest load (this is the same heuristic used for bootstrap). This is rarely called for, as it does not balance the ring in a meaningful way. See *Adding Capacity to an Existing Cluster*.

netstats *host*

Displays network information such as the status of data streaming operations (bootstrap, repair, move and decommission) as well as the number of active, pending and completed commands and responses.

removetoken status | force | *token*

Shows status of a current token removal, forces the the completion of a pending removal, or removes a specified token. This token's range is assumed by another node and the data is streamed there from the remaining live replicas.

See:

- http://wiki.apache.org/cassandra/Operations#Removing_nodes_entirely

setcachecapacity *keyspace cf_name key_cache_capacity row_cache_capacity*

Sets the size of the key cache and row cache. These may be either an absolute number or a percentage in the form of a floating point number.

invalidatekeycache [*keyspace*] [*cfnames*]

Invalidates, or deletes, the key cache. Optionally takes a keyspace or list of column family names (leave a blank space between each column family name).

invalidaterowcache [*keyspace*] [*cfnames*]

Invalidates, or deletes, the row cache. Optionally takes a keyspace or list of column family names (leave a blank space between each column family name).

getcompactionthreshold *keyspace cf_name*

Gets the current compaction threshold settings for a column family. See:

- http://wiki.apache.org/cassandra/MemtableSSTable

setcompactionthreshold *cf_name min_threshold* [*max_threshold*]

The *min_threshold* parameter controls how many SSTables of a similar size must be present before a minor compaction is scheduled. The *max_threshold* sets an upper bound on the number of SSTables that may be compacted in a single minor compaction. See also:

- http://wiki.apache.org/cassandra/MemtableSSTable

cassandra

The `cassandra` utility starts the Cassandra Java server process.

Usage

cassandra [OPTIONS]

Environment

Cassandra requires the following environment variables to be set:

- JAVA_HOME - The path location of your Java Virtual Machine (JVM) installation
- CLASSPATH - A path containing all of the required Java class files (.jar)
- CASSANDRA_CONF - Directory containing the Cassandra configuration files

For convenience, Cassandra uses an include file, `cassandra.in.sh`, to source these environment variables. It will check the following locations for this file:

- Environment setting for CASSANDRA_INCLUDE if set
- $CASSANDRA_HOME/bin
- /usr/share/cassandra/cassandra.in.sh
- /usr/local/share/cassandra/cassandra.in.sh
- /opt/cassandra/cassandra.in.sh
- $HOME/.cassandra.in.sh

Cassandra also uses the Java options set in `$CASSANDRA_CONF/cassandra-env.sh`. If you want to pass additional options to the Java virtual machine, such as maximum and minimum heap size, edit the options in that file rather than setting JVM_OPTS in the environment.

Options

-f

Start the `cassandra` process in foreground (default is to start as a background process).

-p *<filename>*

Log the process ID in the named file. Useful for stopping Cassandra by killing its PID.

-v

Print the version and exit.

-D *<parameter>*

Passes in one of the following startup parameters:

Parameter	Description
`access.properties=<filename>`	The file location of the access.properties file.
`cassandra-pidfile=<filename>`	Log the Cassandra server process ID in the named file. Useful for stopping Cassandra by killing its PID.
`cassandra.config=<directory>`	The directory location of the Cassandra configuration files.
`cassandra.initial_token=<token>`	Sets the initial partitioner token for a node the first time the node is started.
`cassandra.join_ring=<true|false>`	Set to false to start Cassandra on a node but not have the node join the cluster.
`cassandra.load_ring_state=<true|false>`	Set to false to clear all gossip state for the node on restart. Use if you have changed node information in `cassandra.yaml` (such as `listen_address`).
`cassandra.renew_counter_id=<true|false>`	Set to true to reset local counter info on a node. Used to recover from data loss to a counter column family. First remove all SSTables for counter column families on the node, then restart the node with `-Dcassandra.renew_counter_id=true`, then run `nodetool repair` once the node is up again.
`cassandra.replace_token=<token>`	To replace a node that has died, restart a new node in its place and use this parameter to pass in the token that the new node is assuming. The new node must not have any data in its data directory and the token passed must already be a token that is part of the ring.

cassandra.framed cassandra.host cassandra.port=<port> cassandra.rpc_port=<port> cassandra.start_rpc=<true|false> cassandra.storage_port=<port> corrupt-sstable-root legacy-sstable-root mx4jaddress mx4jport passwd.mode passwd.properties=<file>

Examples

Start Cassandra on a node and log its PID to a file:

```
cassandra -p ./cassandra.pid
```

Clear gossip state when starting a node. This is useful if the node has changed its configuration, such as its listen IP address:

```
cassandra -Dcassandra.load_ring_state=false
```

Start Cassandra on a node in stand-alone mode (do not join the cluster configured in the cassandra.yaml file):

```
cassandra -Dcassandra.join_ring=false
```

stress

The `/tools/stress` directory contains the Java-based stress testing utilities that can help in benchmarking and load testing a Cassandra cluster: `stress.java` and the daemon `stressd`. The daemon mode, which keeps the JVM warm more efficiently, may be useful for large-scale benchmarking.

Setting up the Stress Utility

Use Apache ant to to build the stress testing tool:

1. Run `ant` from the Cassandra source directory.
2. Run `ant` from the /tools/stress directory.

Usage

There are three different modes of operation:

- inserting (loading test data)
- reading
- range slicing (only works with the OrderPreservingPartioner)
- indexed range slicing (works with RandomParitioner on indexed ColumnFamiles).

You can use these modes with or without the `stressd` daemon running. For larger-scale testing, the daemon can yield better performance by keeping the JVM warm and preventing potential skew in test results.

If no specific operation is specified, `stress` will insert 1M rows.

The options available are:

-o <operation>, --operation <operation>

Sets the operation mode, one of 'insert', 'read', 'rangeslice', or 'indexedrangeslice'

-T <IP>, --send-to <IP>

Sends the command as a request to the stress daemon at the specified IP address. The daemon must already be running at that address.

-n <NUMKEYS>, --num-keys <NUMKEYS>

Number of keys to write or read. Default is 1,000,000.

-l <RF>, --replication-factor <RF>

Replication Factor to use when creating needed column families. Defaults to 1.

-R <strategy>, --replication-strategy <strategy>

Replication strategy to use (only on insert when keyspace does not exist. Default is:org.apache.cassandra.locator.SimpleStrategy.

-O <properties>, --strategy-properties <properties>

Replication strategy properties in the following format <dc_name>:<num>,<dc_name>:<num>,... Use with network topology strategy.

-W, --no-replicate-on-write

Set replicate_on_write to false for counters. Only for counters add with CL=ONE.

-e <CL>, --consistency-level <CL>

Consistency Level to use (ONE, QUORUM, LOCAL_QUORUM, EACH_QUORUM, ALL, ANY). Default is ONE.

-c <COLUMNS>, --columns <COLUMNS>

Number of columns per key. Default is 5.

-d <NODES>, --nodes <NODES>

Nodes to perform the test against.(comma separated, no spaces). Default is “localhost”.

-y <TYPE>, --family-type <TYPE>

Sets the ColumnFamily type. One of 'Standard' or 'Super'. If using super, set the -u option also.

-V, --average-size-values

Generate column values of average rather than specific size.

-u <SUPERCOLUMNS>, --supercolumns <SUPERCOLUMNS>

Use the number of supercolumns specified. You must set the -y option appropriately, or this option has no effect.

-g <COUNT>, --get-range-slice-count <COUNT>

Sets the number of rows to slice at a time and defaults to 1000. This is only used for the rangeslice operation and will *NOT* work with the RandomPartioner. You must set the OrderPreservingPartioner in your storage configuration (note that you will need to wipe all existing data when switching partioners.)

-g <KEYS>, --keys-per-call <KEYS>

Number of keys to get_range_slices or multiget per call. Default is 1000.

-r, --random

Only used for reads. By default, stress will perform reads on rows with a Guassian distribution, which will cause some repeats. Setting this option makes the reads completely random instead.

-i, --progress-interval

The interval, in seconds, at which progress will be output.

Using the Daemon Mode (`stressd`)

Usage for the daemon mode is:

```
/tools/stress/bin/stressd start|stop|status [-h <host>]
```

During stress testing, you can keep the daemon running and send `stress.java` commands through it using the `-T` or `--send-to` option flag.

Examples

1M inserts to given host:

```
/tools/stress/bin/stress -d 192.168.1.101
```

1M reads from given host:

```
tools/stress/bin/stress -d 192.168.1.101 -o read
```

10M inserts spread across two nodes:

```
/tools/stress/bin/stress -d 192.168.1.101,192.168.1.102 -n 10000000
```

10M inserts spread across two nodes using the daemon mode:

```
/tools/stress/bin/stress -d 192.168.1.101,192.168.1.102 -n 10000000 -T 54.0.0.1
```

sstable2json / json2sstable

The utility `sstable2json` converts the on-disk SSTable representation of a column family into a JSON formatted document. Its counterpart, `json2sstable` , does exactly the opposite: it converts a JSON representation of a column family to a Cassandra usable SSTable format. Converting SSTables this way can be useful for testing and debugging.

Note

Starting with version 0.7, `json2sstable` and `sstable2json` must be run in such a way that the schema can be loaded from system tables. This means that `cassandra.yaml` must be found in the classpath and refer to valid storage directories.

See also: The Import/Export section of http://wiki.apache.org/cassandra/Operations.

sstable2json

This converts the on-disk SSTable representation of a column family into a JSON formatted document.

Usage

```
bin/sstable2json [-f OUT_FILE] SSTABLE
    [-k KEY [-k KEY [...]]]] [-x KEY [-x KEY [...]]] [-e]
```

`SSTABLE` should be a full path to a *`column-family-name`*`-Data.db` file in Cassandra’s data directory. For example, `/var/lib/cassandra/data/Keyspace1/Standard1-e-1-Data.db`.

`-k` allows you to include a specific set of keys. Limited to 500 keys.

`-x` allows you to exclude a specific set of keys. Limited to 500 keys.

`-e` causes keys to only be enumerated

Output Format

The output of `sstable2json` for standard column families is:

```
{
  ROW_KEY:
  {
    [
      [COLUMN_NAME, COLUMN_VALUE, COLUMN_TIMESTAMP, IS_MARKED_FOR_DELETE],
      [COLUMN_NAME, ... ],
      ...
    ]
  },
  ROW_KEY:
  {
    ...
  },
  ...
}
```

The output for super column families is:

```
{
  ROW_KEY:
  {
    SUPERCOLUMN_NAME:
    {
      deletedAt: DELETION_TIME,
      subcolumns:
      [
        [COLUMN_NAME, COLUMN_VALUE, COLUMN_TIMESTAMP, IS_MARKED_FOR_DELETE],
        [COLUMN_NAME, ... ],
        ...
      ]
    },
    SUPERCOLUMN_NAME:
    {
      ...
    },
    ...
  },
  ROW_KEY:
  {
    ...
  },
  ...
}
```

Row keys, column names and values are written in as the hex representation of their byte arrays. Line breaks are only in between row keys in the actual output.

json2sstable

This converts a JSON representation of a column family to a Cassandra usable SSTable format.

Usage

```
bin/json2sstable -K KEYSPACE -c COLUMN_FAMILY JSON SSTABLE
```

`JSON` should be a path to the JSON file

`SSTABLE` should be a full path to a *column-family-name*-Data.db file in Cassandra’s data directory. For example, `/var/lib/cassandra/data/Keyspace1/Standard1-e-1-Data.db`.

sstablekeys

The `sstablekeys` utility is shorthand for `sstable2json` with the `-e` option. Instead of dumping all of a column family’s data, it dumps only the keys.

Usage

```
bin/sstablekeys SSTABLE
```

`SSTABLE` should be a full path to a *column-family-name*-Data.db file in Cassandra’s data directory. For example, `/var/lib/cassandra/data/Keyspace1/Standard1-e-1-Data.db`.

Troubleshooting Guide

This page contains recommended fixes and workarounds for issues commonly encountered with Cassandra.

Reads are getting slower while writes are still fast

Check the SSTable counts in *cfstats*. If the count is continually growing, the cluster's IO capacity is not enough to handle the write load it is receiving. Reads have slowed down because the data is fragmented across many SSTables and compaction is continually running trying to reduce them. Adding more IO capacity, either via more machines in the cluster, or faster drives such as SSDs, will be necessary to solve this.

If the SSTable count is relatively low (32 or less) then the amount of file cache available per machine compared to the amount of data per machine needs to be considered, as well as the application's read pattern. The amount of file cache can be formulated as (TotalMemory – JVMHeapSize) and if the amount of data is greater and the read pattern is approximately random, an equal ratio of reads to the cache:data ratio will need to seek the disk. With spinning media, this is a slow operation. You may be able to mitigate many of the seeks by using a key cache of 100%, and a small amount of row cache (10000-20000) if you have some 'hot' rows and they are not extremely large.

Nodes seem to freeze after some period of time

Check your system.log for messages from the GCInspector. If the GCInspector is indicating that either the ParNew or ConcurrentMarkSweep collectors took longer than 15 seconds, there is a very high probability that some portion of the JVM is being swapped out by the OS. One way this might happen is if the mmap DiskAccessMode is used without JNA support. The address space will be exhausted by mmap, and the OS will decide to swap out some portion of the JVM that isn't in use, but eventually the JVM will try to GC this space. Adding the JNA libraries will solve this (they cannot be shipped with Cassandra due to carrying a GPL license, but are freely available) or the DiskAccessMode can be switched to mmap_index_only, which as the name implies will only mmap the indicies, using much less address space. DataStax recommends that Cassandra nodes disable swap entirely, since it is better to have the OS OutOfMemory (OOM) killer kill the Java process entirely than it is to have the JVM buried in swap and responding poorly.

If the GCInspector isn't reporting very long GC times, but is reporting moderate times frequently (ConcurrentMarkSweep taking a few seconds very often) then it is likely that the JVM is experiencing extreme GC pressure and will eventually OOM. See the section below on OOM errors.

Nodes are dying with OOM errors

If nodes are dying with OutOfMemory exceptions, check for these typical causes:

- Row cache is too large, or is caching large rows
  - Row cache is generally a high-end optimization. Try disabling it and see if the OOM problems continue.
- The memtable sizes are too large for the amount of heap allocated to the JVM
  - You can expect N + 2 memtables resident in memory, where N is the number of column families. Adding another 1GB on top of that for Cassandra itself is a good estimate of total heap usage.

If none of these seem to apply to your situation, try loading the heap dump in MAT and see which class is consuming the bulk of the heap for clues.

Nodetool or JMX Connections Failing on Remote Nodes

If you can run nodetool commands locally but not on other nodes in the ring, you may have a common JMX connection problem that is resolved by adding an entry like the following in `$CASSANDRA_HOME/conf/cassandra-env.sh` on each node:

```
JVM_OPTS="$JVM_OPTS -Djava.rmi.server.hostname=<public name>"
```

If you still cannot run nodetool commands remotely after making this configuration change, do a full evaluation of your firewall and network security. The nodetool utility communciates through JMX on port 7199.

View of ring differs between some nodes

This is an indication that the ring is in a bad state. This can happen when there are token conflicts (for instance, when bootstrapping two nodes simultaneously with automatic token selection.) Unfortunately, the only way to resolve this is to do a full cluster restart; a rolling restart is insufficient since gossip from nodes with the bad state will repopulate it on newly booted nodes.

Java reports an error saying there are too many open files

One possibility is that Java is not allowed to open enough file descriptors. Cassandra generally needs more than the default (1024) amount. This can be adjusted by increasing the security limits on your Cassandra nodes. For example, using the following commands:

```
echo "* soft nofile 32768" | sudo tee -a /etc/security/limits.conf
echo "* hard nofile 32768" | sudo tee -a /etc/security/limits.conf
echo "root soft nofile 32768" | sudo tee -a /etc/security/limits.conf
echo "root hard nofile 32768" | sudo tee -a /etc/security/limits.conf
```

Another, much less likely possibility, is a file descriptor leak in Cassandra. See if the number of file descriptors opened by java seems reasonable when running `lsof -n | grep java` and report the error if the number is greater than a few thousand.