SQLite

sql database engine

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain.

This tutorial will give you a quick start with SQLite and make you comfortable with SQLite programming.

Audience

This tutorial has been prepared for beginners to help them understand the basic-to-advanced concepts related to SQLite Database Engine.

Prerequisites

Before you start practicing various types of examples given in this reference, we assume that you are already aware about what is a database, especially RDBMS and what is a computer programming language.

Disclaimer & Copyright

Table of Contents

SQLite Basics

1. SQLite – Overview

This chapter helps you understand what is SQLite, how it differs from SQL, why it is needed and the way in which it handles the applications Database.

SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is one of the fastest-growing database engines around, but that's growth in terms of popularity, not anything to do with its size. The source code for SQLite is in the public domain.

What is SQLite?

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. It is a database, which is zero-configured, which means like other databases you do not need to configure it in your system.

SQLite engine is not a standalone process like other databases, you can link it statically or dynamically as per your requirement with your application. SQLite accesses its storage files directly.

Why SQLite?

- SQLite does not require a separate server process or system to operate (serverless).
- SQLite comes with zero-configuration, which means no setup or administration needed.
- A complete SQLite database is stored in a single cross-platform disk file.
- SQLite is very small and light weight, less than 400KiB fully configured or less than 250KiB with optional features omitted.
- SQLite is self-contained, which means no external dependencies.
- SQLite transactions are fully ACID-compliant, allowing safe access from multiple processes or threads.
- SQLite supports most of the query language features found in SQL92 (SQL2) standard.
- SQLite is written in ANSI-C and provides simple and easy-to-use API.
- SQLite is available on UNIX (Linux, Mac OS-X, Android, iOS) and Windows (Win32, WinCE, WinRT).

SQLite – A Brief History

- 2000 - D. Richard Hipp designed SQLite for the purpose of no administration required for operating a program.
- 2000 - In August, SQLite 1.0 released with GNU Database Manager.
- 2011 - Hipp announced to add UNQl interface to SQLite DB and to develop UNQLite (Document oriented database).

SQLite Limitations

There are few unsupported features of SQL92 in SQLite which are listed in the following table.

Feature	Description
RIGHT OUTER JOIN	Only LEFT OUTER JOIN is implemented.
FULL OUTER JOIN	Only LEFT OUTER JOIN is implemented.
ALTER TABLE	The RENAME TABLE and ADD COLUMN variants of the ALTER TABLE command are supported. The DROP COLUMN, ALTER COLUMN, ADD CONSTRAINT are not supported.
Trigger support	FOR EACH ROW triggers are supported but not FOR EACH STATEMENT triggers.
VIEWs	VIEWs in SQLite are read-only. You may not execute a DELETE, INSERT, or UPDATE statement on a view.
GRANT and REVOKE	The only access permissions that can be applied are the normal file access permissions of the underlying operating system.

SQLite Commands

The standard SQLite commands to interact with relational databases are similar to SQL. They are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into groups based on their operational nature.

DDL - Data Definition Language

Command	Description
CREATE	Creates a new table, a view of a table, or other object in database.

ALTER	Modifies an existing database object, such as a table.
DROP	Deletes an entire table, a view of a table or other object in the database.

DML - Data Manipulation Language

Command	Description
INSERT	Creates a record
UPDATE	Modifies records
DELETE	Deletes records

DQL - Data Query Language

Command	Description
SELECT	Retrieves certain records from one or more tables

2. SQLite – Installation

SQLite is famous for its great feature zero-configuration, which means no complex setup or administration is needed. This chapter will take you through the process of setting up SQLite on Windows, Linux and Mac OS X.

Install SQLite on Windows

Step 1: Go to SQLite download page, and download precompiled binaries from Windows section.

Step 2: Download sqlite-shell-win32-*.zip and sqlite-dll-win32-*.zip zipped files.

Step 3: Create a folder C:\>sqlite and unzip above two zipped files in this folder, which will give you sqlite3.def, sqlite3.dll and sqlite3.exe files.

Step 4: Add C:\>sqlite in your PATH environment variable and finally go to the command prompt and issue sqlite3 command, which should display the following result.

```
C:\>sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Install SQLite on Linux

Today, almost all the flavors of Linux OS are being shipped with SQLite. So you just issue the following command to check if you already have SQLite installed on your machine.

```
$sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

If you do not see the above result, then it means you do not have SQLite installed on your Linux machine. Following are the steps to install SQLite:

Step 1: Go to SQLite download page and download sqlite-autoconf-*.tar.gz from source code section.

Step 2: Run the following command.

```
$tar xvfz sqlite-autoconf-3071502.tar.gz
$cd sqlite-autoconf-3071502
$./configure --prefix=/usr/local
$make
$make install
```

The above command will end with SQLite installation on your Linux machine, which you can verify as explained above.

Install SQLite on Mac OS X

Though the latest version of Mac OS X comes pre-installed with SQLite, but if you do not have installation available then just follow these steps:

Step 1: Go to SQLite download page, and download sqlite-autoconf-*.tar.gz from source code section.

Step 2: Run the following command.

```
$tar xvfz sqlite-autoconf-3071502.tar.gz
$cd sqlite-autoconf-3071502
$./configure --prefix=/usr/local
$make
$make install
```

The above procedure will end with SQLite installation on your Mac OS X machine, which you can verify by issuing the following command:

```
$sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Finally, you have SQLite command prompt where you can issue SQLite commands for your exercises.

3. SQLite – Commands

This chapter will take you through simple and useful commands used by SQLite programmers. These commands are called SQLite dot commands and exception with these commands is that they should not be terminated by a semi-colon (;).

Let's start with typing a simple **sqlite3** command at command prompt which will provide you with SQLite command prompt where you will issue various SQLite commands.

```
$sqlite3
SQLite version 3.3.6
Enter ".help" for instructions
sqlite>
```

For a listing of the available dot commands, you can enter ".help" any time. For example:

```
sqlite>.help
```

The above command will display a list of various important SQLite dot commands, which are listed in the following table.

Command	Description
.backup ?DB? FILE	Backup DB (default "main") to FILE
.bail ON|OFF	Stop after hitting an error. Default OFF
.databases	List names and files of attached databases
.dump ?TABLE?	Dump the database in an SQL text format. If TABLE specified, only dump tables matching LIKE pattern TABLE
.echo ON|OFF	Turn command echo on or off
.exit	Exit SQLite prompt
.explain ON|OFF	Turn output mode suitable for EXPLAIN on or off. With no args, it turns EXPLAIN on
.header(s) ON|OFF	Turn display of headers on or off
.help	Show this message

.import FILE TABLE	Import data from FILE into TABLE
.indices ?TABLE?	Show names of all indices. If TABLE specified, only show indices for tables matching LIKE pattern TABLE
.load FILE ?ENTRY?	Load an extension library
.log FILE|off	Turn logging on or off. FILE can be stderr/stdout
.mode MODE	Set output mode where MODE is one of: • **csv** Comma-separated values • **column** Left-aligned columns • **html** HTML <table> code • **insert** SQL insert statements for TABLE • **line** One value per line • **list** Values delimited by .separator string • **tabs** Tab-separated values • **tcl** TCL list elements
.nullvalue STRING	Print STRING in place of NULL values
.output FILENAME	Send output to FILENAME
.output stdout	Send output to the screen
.print STRING...	Print literal STRING
.prompt MAIN CONTINUE	Replace the standard prompts
.quit	Exit SQLite prompt

.read FILENAME	Execute SQL in FILENAME
.schema ?TABLE?	Show the CREATE statements. If TABLE specified, only show tables matching LIKE pattern TABLE
.separator STRING	Change separator used by output mode and .import
.show	Show the current values for various settings
.stats ON|OFF	Turn stats on or off
.tables ?PATTERN?	List names of tables matching a LIKE pattern
.timeout MS	Try opening locked tables for MS milliseconds
.width NUM NUM	Set column widths for "column" mode
.timer ON|OFF	Turn the CPU timer measurement on or off

Let's try **.show** command to see default setting for your SQLite command prompt.

```
sqlite>.show
     echo: off
  explain: off
  headers: off
     mode: column
nullvalue: ""
   output: stdout
separator: "|"
    width:
sqlite>
```

Make sure there is no space in between sqlite> prompt and dot command, otherwise it will not work.

Formatting Output

You can use the following sequence of dot commands to format your output.

```
sqlite>.header on
sqlite>.mode column
sqlite>.timer on
sqlite>
```

The above setting will produce the output in the following format.

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
CPU Time: user 0.000000 sys 0.000000
```

The sqlite_master Table

The master table holds the key information about your database tables and it is called **sqlite_master**. You can see its schema as follows:

```
sqlite>.schema sqlite_master
```

This will produce the following result.

```
CREATE TABLE sqlite_master (
  type text,
  name text,
  tbl_name text,
  rootpage integer,
  sql text
```

```
);
```

4. SQLite – Syntax

SQLite is followed by a unique set of rules and guidelines called Syntax. This chapter lists all the basic SQLite Syntax.

Case Sensitivity

Important point to be noted is that SQLite is **case insensitive**, but there are some commands, which are case sensitive like **GLOB** and **glob** have different meaning in SQLite statements.

Comments

SQLite comments are extra notes, which you can add in your SQLite code to increase its readability and they can appear anywhere; whitespace can occur, including inside expressions and in the middle of other SQL statements but they cannot be nested.

SQL comments begin with two consecutive "-" characters (ASCII 0x2d) and extend up to and including the next newline character (ASCII 0x0a) or until the end of input, whichever comes first.

You can also use C-style comments, which begin with "/*" and extend up to and including the next "*/" character pair or until the end of input, whichever comes first. C-style comments can span multiple lines.

```
sqlite>.help -- This is a single line comment
```

SQLite Statements

All the SQLite statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, etc., and all the statements end with a semicolon (;).

SQLite ANALYZE Statement

```
ANALYZE;

or

ANALYZE database_name;

or

ANALYZE database_name.table_name;
```

SQLite AND/OR Clause

```
SELECT column1, column2....columnN

FROM   table_name
```

```
WHERE  CONDITION-1 {AND|OR} CONDITION-2;
```

SQLite ALTER TABLE Statement

```
ALTER TABLE table_name ADD COLUMN column_def...;
```

SQLite ALTER TABLE Statement (Rename)

```
ALTER TABLE table_name RENAME TO new_table_name;
```

SQLite ATTACH DATABASE Statement

```
ATTACH DATABASE 'DatabaseName' As 'Alias-Name';
```

SQLite BEGIN TRANSACTION Statement

```
BEGIN;
or
BEGIN EXCLUSIVE TRANSACTION;
```

SQLite BETWEEN Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name BETWEEN val-1 AND val-2;
```

SQLite COMMIT Statement

```
COMMIT;
```

SQLite CREATE INDEX Statement

```
CREATE INDEX index_name
ON table_name ( column_name COLLATE NOCASE );
```

SQLite CREATE UNIQUE INDEX Statement

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...columnN);
```

SQLite CREATE TABLE Statement

```
CREATE TABLE table_name(
   column1 datatype,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
   PRIMARY KEY( one or more columns )
);
```

SQLite CREATE TRIGGER Statement

```
CREATE TRIGGER database_name.trigger_name
BEFORE INSERT ON table_name FOR EACH ROW
BEGIN
   stmt1;
   stmt2;
   ....
END;
```

SQLite CREATE VIEW Statement

```
CREATE VIEW database_name.view_name  AS
SELECT statement....;
```

SQLite CREATE VIRTUAL TABLE Statement

```
CREATE VIRTUAL TABLE database_name.table_name USING weblog( access.log );
or
CREATE VIRTUAL TABLE database_name.table_name USING fts3( );
```

SQLite COMMIT TRANSACTION Statement

```
COMMIT;
```

SQLite COUNT Clause

```
SELECT COUNT(column_name)
FROM   table_name
WHERE  CONDITION;
```

SQLite DELETE Statement

```
DELETE FROM table_name
WHERE  {CONDITION};
```

SQLite DETACH DATABASE Statement

```
DETACH DATABASE 'Alias-Name';
```

SQLite DISTINCT Clause

```
SELECT DISTINCT column1, column2....columnN
FROM   table_name;
```

SQLite DROP INDEX Statement

```
DROP INDEX database_name.index_name;
```

SQLite DROP TABLE Statement

```
DROP TABLE database_name.table_name;
```

SQLite DROP VIEW Statement

```
DROP INDEX database_name.view_name;
```

SQLite DROP TRIGGER Statement

```
DROP INDEX database_name.trigger_name;
```

SQLite EXISTS Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name EXISTS (SELECT * FROM   table_name );
```

SQLite EXPLAIN Statement

```
EXPLAIN INSERT statement...;
```

```
or
EXPLAIN QUERY PLAN SELECT statement...;
```

SQLite GLOB Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name GLOB { PATTERN };
```

SQLite GROUP BY Clause

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name;
```

SQLite HAVING Clause

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name
HAVING (arithematic function condition);
```

SQLite INSERT INTO Statement

```
INSERT INTO table_name( column1, column2....columnN)
VALUES ( value1, value2....valueN);
```

SQLite IN Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name IN (val-1, val-2,...val-N);
```

SQLite Like Clause

```
SELECT column1, column2....columnN
FROM   table_name
```

```
WHERE  column_name LIKE { PATTERN };
```

SQLite NOT IN Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name NOT IN (val-1, val-2,...val-N);
```

SQLite ORDER BY Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION
ORDER BY column_name {ASC|DESC};
```

SQLite PRAGMA Statement

```
PRAGMA pragma_name;

For example:

PRAGMA page_size;
PRAGMA cache_size = 1024;
PRAGMA table_info(table_name);
```

SQLite RELEASE SAVEPOINT Statement

```
RELEASE savepoint_name;
```

SQLite REINDEX Statement

```
REINDEX collation_name;
REINDEX database_name.index_name;
REINDEX database_name.table_name;
```

SQLite ROLLBACK Statement

```
ROLLBACK;
or

ROLLBACK TO SAVEPOINT savepoint_name;
```

SQLite SAVEPOINT Statement

```
SAVEPOINT savepoint_name;
```

SQLite SELECT Statement

```
SELECT column1, column2....columnN
FROM   table_name;
```

SQLite UPDATE Statement

```
UPDATE table_name
SET column1 = value1, column2 = value2....columnN=valueN
[ WHERE  CONDITION ];
```

SQLite VACUUM Statement

```
VACUUM;
```

SQLite WHERE Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION;
```

5. SQLite – Data Type

SQLite data type is an attribute that specifies the type of data of any object. Each column, variable and expression has related data type in SQLite.

You would use these data types while creating your tables. SQLite uses a more general dynamic type system. In SQLite, the datatype of a value is associated with the value itself, not with its container.

SQLite Storage Classes

Each value stored in an SQLite database has one of the following storage classes:

Storage Class	Description
NULL	The value is a NULL value.
INTEGER	The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
REAL	The value is a floating point value, stored as an 8-byte IEEE floating point number.
TEXT	The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE)
BLOB	The value is a blob of data, stored exactly as it was input.

SQLite storage class is slightly more general than a datatype. The INTEGER storage class, for example, includes 6 different integer datatypes of different lengths.

SQLite Affinity Type

SQLite supports the concept of **type affinity** on columns. Any column can still store any type of data but the preferred storage class for a column is called its **affinity**. Each table column in an SQLite3 database is assigned one of the following type affinities:

Affinity	Description
TEXT	This column stores all data using storage classes NULL, TEXT or BLOB.
NUMERIC	This column may contain values using all five storage classes.

INTEGER	Behaves the same as a column with NUMERIC affinity, with an exception in a CAST expression.
REAL	Behaves like a column with NUMERIC affinity except that it forces integer values into floating point representation.
NONE	A column with affinity NONE does not prefer one storage class over another and no attempt is made to coerce data from one storage class into another.

SQLite Affinity and Type Names

Following table lists down various data type names which can be used while creating SQLite3 tables with the corresponding applied affinity.

Data Type	Affinity
• INT • INTEGER • TINYINT • SMALLINT • MEDIUMINT • BIGINT • UNSIGNED BIG INT • INT2 • INT8	INTEGER
• CHARACTER(20) • VARCHAR(255) • VARYING CHARACTER(255) • NCHAR(55) • NATIVE CHARACTER(70) • NVARCHAR(100) • TEXT • CLOB	TEXT

• BLOB • no datatype specified	NONE
• REAL • DOUBLE • DOUBLE PRECISION • FLOAT	REAL
• NUMERIC • DECIMAL(10,5) • BOOLEAN • DATE • DATETIME	NUMERIC

Boolean Datatype

SQLite does not have a separate Boolean storage class. Instead, Boolean values are stored as integers 0 (false) and 1 (true).

Date and Time Datatype

SQLite does not have a separate storage class for storing dates and/or times, but SQLite is capable of storing dates and times as TEXT, REAL or INTEGER values.

Storage Class	Date Format
TEXT	A date in a format like "YYYY-MM-DD HH:MM:SS.SSS"
REAL	The number of days since noon in Greenwich on November 24, 4714 B.C.
INTEGER	The number of seconds since 1970-01-01 00:00:00 UTC

You can choose to store dates and times in any of these formats and freely convert between formats using the built-in date and time functions.

End of ebook preview

If you liked what you saw…

Buy it from our store @ https://store.tutorialspoint.com