

# SQLITE QUICK GUIDE

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 the one database, which is zero-configured, that means like other database 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. The 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 the SQL92 *SQL2* standard.
- SQLite is written in ANSI-C and provides simple and easy to use API.
- SQLite is available on UNIX *Linux, MacOS – X, Android, iOS* and Windows *Win32, WinCE, WinRT*.

## History:

1. 2000 -- D. Richard Hipp had designed SQLite for the purpose of no administration required for operating a program.
2. 2000 -- In August, SQLite 1.0 released with GNU Database Manager.
3. 2011 -- Hipp announced to add UNQL interface to SQLite DB and to develop UNQLite *Document oriented database*.

## SQLITE - INSTALLATION

The 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

- Go to [SQLite download page](#), and download precompiled binaries from Windows section.
- You will need to download **sqlite-shell-win32-\*.zip** and **sqlite-dll-win32-\*.zip** zipped files.
- 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.

- Add C:\>sqlite in your PATH environment variable and finally go to the command prompt and issue **sqlite3** command which should display a result something as below.

```
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 flavours 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 or not.

```
$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 above result, then it means you do not have SQLite installed on your Linux machine. So let's follow the following steps to install SQLite:

- Go to [SQLite download page](#) and download **sqlite-autoconf-\*.tar.gz** from source code section.
- Follow the following steps:

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

Above procedure will end with SQLite installation on your Linux machine which you can verify as explained above.

## Install SQLite On Mac OS X

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

- Go to [SQLite download page](#), and download **sqlite-autoconf-\*.tar.gz** from source code section.
- Follow the following steps:

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

Above procedure will end with SQLite installation on your Mac OS X machine which you can verify by issuing 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 to do your

exercises.

## SQLITE - COMMANDS

Let's start with typing a simple **sqlite3** command at command prompt which will provide you 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" at any time. For example:

```
sqlite>.help
```

Above command will display a list of various important SQLite dot commands, which are as follows:

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.
.headers 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: <ul style="list-style-type: none"><li>• <b>csv</b> Comma-separated values</li><li>• <b>column</b> Left-aligned columns.</li><li>• <b>html</b> HTML &lt;table&gt; code</li><li>• <b>insert</b> SQL insert statements for TABLE</li><li>• <b>line</b> One value per line</li><li>• <b>list</b> Values delimited by .separator string</li></ul>

	<ul style="list-style-type: none"> <li>• <b>tabs</b> Tab-separated values</li> <li>• <b>tcl</b> TCL list elements</li> </ul>
.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

## SQLITE - SYNTAX

SQLite is followed by unique set of rules and guidelines called Syntax. This tutorial gives you a quick start with SQLite by listing all the basic SQLite Syntax:

### Case Sensitivity

Important point to be noted is that SQLite is **case insensitive** but there is some command which is 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 can not 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 (arithmetic 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;
```

SQLite data type is an attribute that specifies 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, and corresponding applied affinity also has been shown:

Data Type	Affinity
-----------	----------

<ul style="list-style-type: none"> <li>• INT</li> <li>• INTEGER</li> <li>• TINYINT</li> <li>• SMALLINT</li> <li>• MEDIUMINT</li> <li>• BIGINT</li> <li>• UNSIGNED BIG INT</li> <li>• INT2</li> <li>• INT8</li> </ul>	<p>INTEGER</p>
<ul style="list-style-type: none"> <li>• CHARACTER20</li> <li>• VARCHAR255</li> <li>• VARYING CHARACTER255</li> <li>• NCHAR55</li> <li>• NATIVE CHARACTER70</li> <li>• NVARCHAR100</li> <li>• TEXT</li> <li>• CLOB</li> </ul>	<p>TEXT</p>
<ul style="list-style-type: none"> <li>• BLOB</li> <li>• no datatype specified</li> </ul>	<p>NONE</p>
<ul style="list-style-type: none"> <li>• REAL</li> <li>• DOUBLE</li> <li>• DOUBLE PRECISION</li> <li>• FLOAT</li> </ul>	<p>REAL</p>
<ul style="list-style-type: none"> <li>• NUMERIC</li> <li>• DECIMAL10,5</li> <li>• BOOLEAN</li> <li>• DATE</li> <li>• DATETIME</li> </ul>	<p>NUMERIC</p>

## 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 are capable of storing dates and times as TEXT, REAL, or INTEGER values:

Storage Class	Date Formate
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.

## SQLITE - CREATE DATABASE

The SQLite **sqlite3** command is used to create new SQLite database. You do not need to have any special privilege to create a database.

### Syntax:

Basic syntax of sqlite3 command is as follows:

```
$sqlite3 DatabaseName.db
```

Always database name should be unique within the RDBMS.

## SQLITE - ATTACH DATABASE

Consider a case when you have multiple databases available and you want to use any one of them at a time. SQLite **ATTACH DATABASE** statement is used to select a particular database and after this command, all SQLite statements will be executed under the attached database.

### Syntax:

Basic syntax of SQLite ATTACH DATABASE statement is as follows:

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

Above command will also create a database in case database is already not created, otherwise it will just attach database file name with logical database 'Alias-Name'.

## SQLITE - DETACH DATABASE

SQLite **DETACH DATABASE** statement is used to detach and dissociates a named database from a database connection which was previously attached using ATTACH statement. If the same database file has been attached with multiple aliases, then DETACH command will disconnect only given name and rest of the attachment will still continue. You cannot detach the **main** or **temp** databases.

*If the database is an in-memory or temporary database, the database will be destroyed and the contents will be lost.*

## Syntax:

Basic syntax of SQLite DETACH DATABASE 'Alias-Name' statement is as follows:

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

Here 'Alias-Name' is the same alias which you had used while attaching database using ATTACH statement.

## SQLITE - CREATE TABLE

The SQLite **CREATE TABLE** statement is used to create a new table in any of the given database. Creating a basic table involves naming the table and defining its columns and each column's data type.

## Syntax:

Basic syntax of CREATE TABLE statement is as follows:

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

CREATE TABLE is the keyword telling the database system to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement. Optionally you can specify *database\_name* alongwith *table\_name*.

## SQLITE - DROP TABLE

The SQLite **DROP TABLE** statement is used to remove a table definition and all associated data, indexes, triggers, constraints, and permission specifications for that table.

*You have to be careful while using this command because once a table is deleted then all the information available in the table would also be lost forever.*

## Syntax:

Basic syntax of DROP TABLE statement is as follows. You can optionally specify database name along with table name as follows:

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

## SQLITE - INSERT QUERY

The SQLite **INSERT INTO** Statement is used to add new rows of data into a table in the database.

## Syntax:

There are two basic syntax of INSERT INTO statement is as follows:

```
INSERT INTO TABLE_NAME (column1, column2, column3, ...columnN)]  
VALUES (value1, value2, value3, ...valueN);
```

Here column1, column2,...columnN are the names of the columns in the table into which you want

to insert data.

You may not need to specify the columns name in the SQLite query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table. The SQLite INSERT INTO syntax would be as follows:

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

## SQLITE - SELECT QUERY

SQLite **SELECT** statement is used to fetch the data from a SQLite database table which returns data in the form of result table. These result tables are also called result-sets.

### Syntax:

The basic syntax of SQLite SELECT statement is as follows:

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

Here column1, column2...are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field then you can use following syntax:

```
SELECT * FROM table_name;
```

## SQLITE - OPERATORS

An operator is a reserved word or a character used primarily in an SQLite statement's WHERE clause to perform operations, such as comparisons and arithmetic operations.

Operators are used to specify conditions in an SQLite statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Bitwise operators

### SQLite Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20 then:

[Show Examples](#)

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns	b % a will give 0

remainder

## SQLite Comparison Operators:

Assume variable a holds 10 and variable b holds 20 then:

[Show Examples](#)

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	a == b is not true.
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	a = b is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	a != b is true.
<>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	a <> b is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	a > b is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	a < b is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	a >= b is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	a <= b is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	a !< b is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	a !> b is true.

## SQLite Logical Operators:

Here is a list of all the logical operators available in SQLite.

[Show Examples](#)

Operator	Description
AND	The AND operator allows the existence of multiple conditions in an SQL

	statement's WHERE clause.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
NOT IN	The negation of IN operator which is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
GLOB	The GLOB operator is used to compare a value to similar values using wildcard operators. Also, GLOB is case sensitive, unlike LIKE.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg. NOT EXISTS, NOT BETWEEN, NOT IN etc. <b>This is negate operator.</b>
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
IS	The IS operator work like =
IS NOT	The IS operator work like !=
	Adds two different string and make new one.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness no duplicates.

## SQLite Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation. The truth table for & and | is as follows:

p	q	p & q	p   q
0	0	0	0
0	1	0	1
1	1	1	1
1	0	0	1

Assume if A = 60; and B = 13; Now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

-----

A&B = 0000 1100

A|B = 0011 1101

~A = 1100 0011

The Bitwise operators supported by SQLite language are listed in the following table. Assume variable A holds 60 and variable B holds 13 then:

[Show Examples](#)

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	A & B will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	A   B will give 61 which is 0011 1101
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	~A will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

## SQLITE - EXPRESSIONS

An expression is a combination of one or more values, operators, and SQL functions that evaluate to a value.

SQL EXPRESSIONS are like formulas and they are written in query language. You can also use to query the database for specific set of data.

### Syntax:

Consider the basic syntax of the SELECT statement as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [CONTION | EXPRESSION];
```

There are different types of SQLite expression, which are mentioned below:

### SQLite - Boolean Expressions:

SQLite Boolean Expressions fetch the data on the basis of matching single value. Following is the syntax:

```
SELECT column1, column2, columnN
FROM table_name
WHERE SINGLE VALUE MATCHTING EXPRESSION;
```

### SQLite - Numeric Expression:

These expressions are used to perform any mathematical operation in any query. Following is the syntax:

```
SELECT numerical_expression as OPERATION_NAME  
[FROM table_name WHERE CONDITION] ;
```

Here numerical\_expression is used for mathematical expression or any formula. Following is a simple examples showing usage of SQL Numeric Expressions:

```
sqlite> SELECT (15 + 6) AS ADDITION  
ADDITION = 21
```

There are several built-in functions like avg, sum, count etc to perform what is known as aggregate data calculations against a table or a specific table column.

```
sqlite> SELECT COUNT(*) AS "RECORDS" FROM COMPANY;  
RECORDS = 7
```

## SQLite - Date Expressions:

Date Expressions return current system date and time values and these expressions will be used in various data manipulation.

```
sqlite> SELECT CURRENT_TIMESTAMP;  
CURRENT_TIMESTAMP = 2013-03-17 10:43:35
```

## SQLITE - WHERE CLAUSE

The SQLite **WHERE** clause is used to specify a condition while fetching the data from one table or multiple tables.

If the given condition is satisfied, means true then it returns specific value from the table. You would use WHERE clause to filter the records and fetching only necessary records.

The WHERE clause not only used in SELECT statement, but it is also used in UPDATE, DELETE statement etc. which we would study in subsequent chapters.

### Syntax:

The basic syntax of SQLite SELECT statement with WHERE clause is as follows:

```
SELECT column1, column2, columnN  
FROM table_name  
WHERE [condition]
```

## SQLITE - AND AND OR OPERATORS

### The AND Operator:

The **AND** operator allows the existence of multiple conditions in an SQLite statement's WHERE clause. While using AND operator, complete condition will be assumed true when all the conditions are true. For example [condition1] AND [condition2] will be true only when both condition1 and condition2 are true.

### Syntax:

The basic syntax of AND operator with WHERE clause is as follows:

```
SELECT column1, column2, columnN  
FROM table_name  
WHERE [condition1] AND [condition2]...AND [conditionN];
```

You can combine N number of conditions using AND operator. For an action to be taken by the SQLite statement, whether it be a transaction or query, all conditions separated by the AND must

be TRUE.

## The OR Operator:

The OR operator is also used to combine multiple conditions in an SQLite statement's WHERE clause. While using OR operator, complete condition will be assumed true when atleast any of the the conditions is true. For example [condition1] OR [condition2] will be true if either condition1 or condition2 is true.

### Syntax:

The basic syntax of OR operator with WHERE clause is as follows:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

You can combine N number of conditions using OR operator. For an action to be taken by the SQLite statement, whether it be a transaction or query, only any ONE of the conditions separated by the OR must be TRUE.

## SQLITE - UPDATE QUERY

The SQLite **UPDATE** Query is used to modify the existing records in a table. You can use WHERE clause with UPDATE query to update selected rows otherwise all the rows would be updated.

### Syntax:

The basic syntax of UPDATE query with WHERE clause is as follows:

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

You can combine N number of conditions using AND or OR operators.

## SQLITE - DELETE QUERY

The SQLite **DELETE** Query is used to delete the existing records from a table. You can use WHERE clause with DELETE query to delete selected rows, otherwise all the records would be deleted.

### Syntax:

The basic syntax of DELETE query with WHERE clause is as follows:

```
DELETE FROM table_name
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

## CONCLUSION

If you enjoyed this short and quick tutorial and interested in reading more then I will recommend you to go through the detailed tutorial because still you are missing many important concepts related to SQLite

Processing math: 12%