SQLite
sql database engine

tutorialspoint
SIMPLYEASYLEARNING

www.tutorialspoint.com

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

© Copyright 2017 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com.

Table of Contents

SQLite Basics

SQL tutorial   3   .

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.

MS access   . MySQL   DB
                            , Oracle
                     DB   .

Why SQLite?   SQLite DBMS

- 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.
  oracle   Java SE Develoment Kit   .

- SQLite transactions are fully ACID-compliant, allowing safe access from multiple processes or threads.

  Atomicity, Consistency, Isolation, Durability

- 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).

UnQLite is an embedded NoSQL (Key/Value store and Document- store) database engine. Unlike most other NoSQL databases, UnQLite does not have a separate server process. UnQLite reads and writes directly to ordinary disk files. A complete database with multiple collections, is contained in a single disk file. The database file format is cross- platform.

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

SQLite   SQL   .
SQL   SQLite
SQLite   .

The standard SQLite com
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.

tutorialspoint
SIMPLYEASYLEARNING

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

TCL (transaction control language): BEGIN, COMMIT, ROLLBACK
DCL(data control language): GRANT, REVOKE

SQLite is famous for its great feature zero-configuration, which means no complex setup or administration is needed. This chapt| sqlite- tools- win32- x86- *.zip SQLite on Windows, Linux and Mac OS   .

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>
```

1. https://github.com/joongyang/Lecture- Materials- for- SQLite/tree/master/ test   testDB.db   .
2.   .
3.   sqlite   .
(db   )   .

(   ) C: Users   joong   Desktop   **sqlite**> sqlite3

4. 3   sqlite   .
5. .quit   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.

tutorialspoint
SIMPLYEASYLEARNING

3. SQLite – Commands

SQLite   . (
          .)   SQLite   . SQLite
                                        . SQLite   SQL
    . 3   SQLite   , 4   SQL   .

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 <u>SQLite dot commands</u>, 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>
```

```
sqlite>  .echo on
sqlite>  .show
sqlite>  .echo off
sqlite>  .show
```

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

** Sqlite   (optional)

```
home>  sqlite3 testDB.db "select * from employees;"
home>  sqlite3 - column - csv testDB.db "select * from department;"
home>  sqlite3 - - help
```

                            output redirection   .

```
home>  sqlite3 - column - header - csv testDB.db "select * from employees;" >  employees.csv
    1          Paul       32        California  20000.0
```

( )   testDB   testDB.sql   backup
        .   .output, .dump, .read   .

2. testDB.sql   .
3. testDB.sql   .
4.   testDB.db   .
5. testDB.sql   testDB.db   .
6.   testDB.db   .

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,
```

( ) scott.db   . (
   scott.db   .)

1)   .
2)   .
3)   . csv   .
4) csv   excel   .
5) scott.db   backup   .

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;
```
TRIGGER

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 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.

binary large object

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	SQL SQL . SQL . SQLite SQL . SQLite storage class . .
• CHARACTER(20) CHAR(20) • VARCHAR(255) • VARYING CHARACTER(255) • NCHAR(55) • NATIVE CHARACTER(70) • NVARCHAR(100) • TEXT • CLOB	char, nchar . char ascii 1 2 . nchar 2 . nchar . nchar, nvarchar . , TEXT
• BLOB • no datatype specified	
• REAL	REAL

character large object block

19

tutorialspoint
SIMPLYEASYLEARNING

• DOUBLE • DOUBLE PRECISION • FLOAT	
• 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

RDBMS   SQL   DATE,
DATETIME   , SQLite

SQLite does not have a separa   e is capable of storing dates and   .

Storage Class	Date Format
TEXT	A date in a format like "YYYY-MM-DD HH:MM:SS.SSS" ISO
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 unix epoch

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.

BC 4713   7980   Julian Period   .
Julian Period   .   BC
4713   11   24   (
)   julian date   .

6. SQLite – CREATE Database

In SQLite, **sqlite3** command is used to create a 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.

Example

If you want to create a new database <testDB.db>, then SQLITE3 statement would be as follows:

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

The above command will create a file **testDB.db** in the current directory. This file will be used as database by SQLite engine. If you have noticed while creating database, sqlite3 command will provide a **sqlite>** prompt after creating a database file successfully.

Once a database is created, you can verify it in the list of databases using the following SQLite **.databases** command.

```
sqlite>.databases
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
```

You will use SQLite **.quit** command to come out of the sqlite prompt as follows:

```
sqlite>.quit
$
```

The .dump Command

You can use **.dump** dot command to export complete database in a text file using the following SQLite command at the command prompt.

```
$sqlite3 testDB.db .dump > testDB.sql
```

The above command will convert the entire contents of **testDB.db** database into SQLite statements and dump it into ASCII text file **testDB.sql**. You can perform restoration from the generated testDB.sql in a simple way as follows:

```
$sqlite3 testDB.db < testDB.sql
```

At this moment your database is empty, so you can try above two procedures once you have few tables and data in your database. For now, let's proceed to the next chapter.

SQLite   sqlite3.exe   .
                            . SQLite   .dump

    .

sqlite> .output dump_filename.sql
sqlite> .dump
sqlite> .output stdout

(   )   .

1. scott.db   scott.db   .
2. HR.db   HR.db   .

7. 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

Following is the basic syntax of SQLite ATTACH DATABASE statement.

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

DB   attach   .
DB   .

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

Example

If you want to attach an existing database **testDB.db**, then ATTACH DATABASE statement would be as follows:

```
sqlite> ATTACH DATABASE 'testDB.db' as 'TEST';
```

Use SQLite **.database** command to display attached database.

```
sqlite> .database
seq  name            file
---  --------------  ----------------------
0    main            /home/sqlite/testDB.db
2    test            /home/sqlite/testDB.db
```

.databases   .

DB   attach
.

The database names **main** and **temp** are reserved for the primary database and database to hold temporary tables and other temporary data objects. Both of these database names exist for every database connection and should not be used for attachment, otherwise you will get the following warning message.

```
sqlite>  ATTACH DATABASE 'testDB.db' as 'TEMP';

Error: database TEMP is already in use

sqlite>  ATTACH DATABASE 'testDB.db' as 'main';

Error: database TEMP is already in use
```

(   ) attach   DB   schema   . alias   'main', 'temp'   DB   schema   .

8. SQLite – DETACH Database

SQLite **DETACH DATABASE** statement is used to detach and dissociate 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 the 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.

> attach database   (   )
>   db   (   )
>   .

Syntax

Following is the basic syntax of SQLite DETACH DATABASE 'Alias-Name' statement.

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

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

Example

Consider you have a database, which you created in the previous chapter and attached it with 'test' and 'currentDB' as we can see using **.database** command.

```
sqlite>.databases
seq  name              file
---  ---------------   ----------------------
0    main              /home/sqlite/testDB.db
2    test              /home/sqlite/testDB.db
3    currentDB         /home/sqlite/testDB.db
```

> db
> (alias)   attach
>   .

> main   db   detach
>   ?

Let's try to detach 'currentDB' from testDB.db using the following command.

```
sqlite> DETACH DATABASE 'currentDB';
```

Now, if you will check the current attachment, you will find that testDB.db is still connected with 'test' and 'main'.

> (1) chinook.db, hr.db, school.db, scott.db   (connection)
>   ,   DB   .   schema
> diagram   .
>
> (2)   SQL
>   .
>
> SELECT * FROM   ;
>
>   DB   DB   .
>   .   .
>
> (3)   DB   .

9. SQLite – CREATE Table

Syntax

Following is the basic syntax of CREATE TABLE statement.

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

RDB
.
.
.

27 storage class, affinity type, SQL
data type   . create table
SQL   data type   .
SQLite   .

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* along with *table_name*.

Example

schema diagram
. testDB schema diagram
.

```
sqlite> CREATE TABLE COMPANY(
    ID INT PRIMARY KEY     NOT NULL,
    NAME          TEXT     NOT NULL,
    AGE           INT      NOT NULL,
    ADDRESS       CHAR(50),
    SALARY        REAL
);
```

(constraint)？

(constraint)？

Let us create one more table, which we will use in our exercises in subsequent chapters.

```
sqlite> CREATE TABLE DEPARTMENT(
    ID INT PRIMARY KEY     NOT NULL,
    DEPT          CHAR(50) NOT NULL,
    EMP_ID        INT      NOT NULL
);
```

.

emp_id int not null references company(id)

25

You can verify if your table has been created successfully using SQLite command **.tables** command, which will be used to list down all the tables in an attached database.

```
sqlite>.tables
COMPANY        DEPARTMENT
```

Here, you can see the COMPANY table twice because its showing COMPANY table for main database and test.COMPANY table for 'test' alias created for your testDB.db. You can get complete information about a table using the following SQLite **.schema** command.

```
sqlite>.schema COMPANY
CREATE TABLE COMPANY(
    ID INT PRIMARY KEY     NOT NULL,
    NAME           TEXT    NOT NULL,
    AGE            INT     NOT NULL,
    ADDRESS        CHAR(50),
    SALARY         REAL
);
```

```
drop table

create table company(
id int not null
, name text not null
, age int not null
, address char(50)
, salary real
, constraint com_pk primary key (id)
);
create table department(
id int not null
, dept char(50) not null
, emp_id int not null
, constraint dept_pk primary key(id)
, constraint dept_com_fk foreign key(emp_id) references company(id)
);
```

drop table DB   ;

constraint

```
sqlite                              .


     ,                           .


sqlite>  pragma foreign_keys;
sqlite>  pragma foreign_keys= on;
sqlite>  pragma foreign_keys= off;
```

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

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

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

Example

Let us first verify COMPANY table and then we will delete it from the database.

```
sqlite>.tables
COMPANY       test.COMPANY
```

This means COMPANY table is available in the database, so let us drop it as follows:

```
sqlite>DROP TABLE COMPANY;
sqlite>
```

Now, if you try **.TABLES** command, then you will not find COMPANY table anymore.

```
sqlite>.tables
sqlite>
```

It shows nothing which means the table from your database has been dropped successfully.

( ) scott.db .

1. scott.db   schema diagram   .
scott.db   schema daiagram
        .
2. scott.db   scott_twin.db   . scott.db
   Schema diagram   .   unicode
                                                        .

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

Syntax

DB .

Following are the two basic syntaxes of INSERT INTO statement.

```
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 column(s) name in the SQLite query if you are adding values for all the columns of the table. However, make su[...]
the same order as the columns in the table. The SQLite INS[...]   create table
follows:

.

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

Example

Consider you already have created COMPANY table in your testDB.db as follows:

```
sqlite> CREATE TABLE COMPANY(
    ID INT PRIMARY KEY     NOT NULL,
    NAME           TEXT    NOT NULL,
    AGE            INT     NOT NULL,
    ADDRESS        CHAR(50),
    SALARY         REAL
);
```

Now, the following statements would create six records in COMPANY table.

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );


INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Allen', 25, 'Texas', 15000.00 );


INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );


INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );


INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'David', 27, 'Texas', 85000.00 );


INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Kim', 22, 'South-Hall', 45000.00 );
```

You can create a record in COMPANY table using the second syntax as follows:

```
INSERT INTO COMPANY VALUES (7, 'James', 24, 'Houston', 10000.00 );
```

All the above statements would create the following records in COMPANY table. In the next chapter, you will learn how to display all these records from a table.

```
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
```

Populate One Table Using Another Table

You can populate data into a table through select statement over another table provided another table has a set of fields, which are required to populate the first table. Here is the syntax:

select

insert into , .

select select

.

```
INSERT INTO first_table_name [(column1, column2, ... columnN)]
    SELECT column1, column2, ...columnN
    FROM second_table_name
    [WHERE condition];
```

( ) testDB.db schema diagram
mytestDB.db department .
. (
.)

```
id      dept        emp_id
-  - - - - - -  - - - - - -
1       IT Billing    1
2       Engineerin  2
3       Finance      7
```

( ) chinook.db schema diagram mychinook
database .

"PSY 8th 4x2= 8" MPEG audio
file K- pop .
, , , .

I LUV IT, 18900, 5510424,
New Faces, 19000, 7689012,

( ) mychinook.db SQL .

1. . " "
chinook.db . ?
2. chinook.db .
3. (trot) ?
4. " "
.

30

tutorialspoint
SIMPLY EASY LEARNING

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

Syntax

Following is the basic syntax of SQLite SELECT statement.

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

```
SELECT * FROM table_name;
```

Example

Consider COMPANY table with the following records:

```
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
```

Following is an example to fetch and display all these records using SELECT statement. Here, the first three commands have been used to set a properly formatted output.

```
sqlite>.header on

sqlite>.mode column

sqlite> SELECT * FROM COMPANY;
```

Finally, you will get the following result.

```
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
```

If you want to fetch only selected fields of COMPANY table, then use the following query:

```
sqlite> SELECT ID, NAME, SALARY FROM COMPANY;
```

The above query will produce the following result.

```
ID          NAME        SALARY

----------  ----------  ----------

1           Paul        20000.0

2           Allen       15000.0

3           Teddy       20000.0

4           Mark        65000.0

5           David       85000.0

6           Kim         45000.0

7           James       10000.0
```

Setting Output Column Width

sqlite   column   (.mode column)   10   .

                                      .   .width

                        .

```
sqlite>.width 10, 20, 10
sqlite>SELECT * FROM COMPANY;
```

The above **.width** command sets the first column width to 10, the second column width to 20 and the third column width to 10. Finally, the above SELECT statement will give the following result.

```
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
```

Schema Information

As all the **dot commands** are available at SQLite prompt, hence while programming with SQLite, you will use the following SELECT statement with **sqlite_master** table to list down all the tables created in your database.

```
sqlite> SELECT tbl_name FROM sqlite_master WHERE type = 'table';
```

Assuming you have only COMPANY table in your testDB.db, this will produce the following result.

```
tbl_name

----------

COMPANY
```

You can list down complete information about COMPANY table as follows:

```
sqlite> SELECT  sql  FROM  sqlite_master  WHERE  type  =  'table'  AND  tbl_name  =
'COMPANY';
```

What is an Operator in SQLite?

An operator is a reserved word or a character used primarily in an SQLite expression WHERE clause to perform operation(s), 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 SQLite arithmetic operators will be used as follows:

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

SQLite - Arithmetic Operators Example

Following are some simple examples showing the usage of SQLite Arithmetic Operators.

```
sqlite> .mode line
sqlite> select 10 + 20;
```

```
10 + 20 = 30



sqlite> select 10 - 20;
10 - 20 = -10



sqlite> select 10 * 20;
10 * 20 = 200
```

/   .

select 2./3;
select 2/3;

```
sqlite> select 10 / 5;
10 / 5 = 2
```

%

```
sqlite> select 12 %  5;
12 %  5 = 2
```

SQLite Comparison Operators   ←   .

Assume variable **a** holds 10 and variable **b** holds 20, then SQLite comparison operators will be used as follows:

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

35

>=	Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes then the condition becomes true.	(a >= b) is not true.
<=	Checks if the value of the left operand is less than or equal to the value of the right operand, if yes then the condition becomes true.	(a <= b) is true.
!<	Checks if the value of the left operand is not less than the value of the right operand, if yes then the condition becomes true.	(a !< b) is false.
!>	Checks if the value of the left operand is not greater than the value of the right operand, if yes then the condition becomes true.	(a !> b) is true.

SQLite - Comparison Operators Example

Consider COMPANY table with the following records.

```
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
```

select   : from   ,
                    where   ,
                                  .

     select   (
)   , where   .

```
sqlite> SELECT * FROM COMPANY WHERE SALARY > 50000;

ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

4           Mark        25          Rich-Mond   65000.0

5           David       27          Texas       85000.0
```

(   ) select name, salary, salary + 100 from company where salary > 50000;

Following SELECT statement lists down all the records having SALARY equal to 20,000.00.

```
sqlite>  SELECT * FROM COMPANY WHERE SALARY = 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
```

Following SELECT statement lists down all the records having SALARY not equal to 20,000.00.

```
sqlite>  SELECT * FROM COMPANY WHERE SALARY != 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
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
```

Following SELECT statement lists down all the records having SALARY not equal to 20,000.00.

```
sqlite> SELECT * FROM COMPANY WHERE SALARY <> 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
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
```

Following SELECT statement lists down all the records having SALARY greater than or equal to 65,000.00.

```
sqlite> SELECT * FROM COMPANY WHERE SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

SQLite Logical Operators

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

Operator	Description
AND	The AND operator allows the existence of multiple conditions in a 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 a 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 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. Example, NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
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 strings and make a new one.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

tutorialspoint
SIMPLYEASYLEARNING

SQLite - Logical Operators Example

Consider COMPANY table with the following records.

```
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
```

Following are simple examples showing the usage of SQLite Logical Operators. Following SELECT statement lists down all the records where AGE is greater than or equal to 25 and salary is greater than or equal to 65000.00.

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

4           Mark        25          Rich-Mond   65000.0

5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE is greater than or equal to 25 **OR** salary is greater than or equal to 65000.00.

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

1           Paul        32          California  20000.0

2           Allen       25          Texas       15000.0

4           Mark        25          Rich-Mond   65000.0

5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE is not NULL, which means all the records because none of the record has AGE equal to NULL.

```
sqlite>  SELECT * FROM COMPANY WHERE AGE IS NOT NULL;
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
```

Following SELECT statement lists down all the records where NAME starts with 'Ki', does not matter what comes after 'Ki'.

```
sqlite> SELECT * FROM COMPANY WHERE NAME LIKE 'Ki%';

ID           NAME        AGE         ADDRESS     SALARY

----------   ----------  ----------  ----------  ----------

6            Kim         22          South-Hall  45000.0
```

Following SELECT statement lists down all the records where NAME starts with 'Ki', does not matter what comes after 'Ki'.

```
sqlite> SELECT * FROM COMPANY WHERE NAME GLOB 'Ki*';

ID           NAME        AGE         ADDRESS     SALARY

----------   ----------  ----------  ----------  ----------

6            Kim         22          South-Hall  45000.0
```

Following SELECT statement lists down all the records where AGE value is either 25 or 27.

```
sqlite> SELECT * FROM COMPANY WHERE AGE IN ( 25, 27 );

ID           NAME        AGE         ADDRESS     SALARY

----------   ----------  ----------  ----------  ----------

2            Allen       25          Texas       15000.0

4            Mark        25          Rich-Mond   65000.0

5            David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE value is neither 25 nor 27.

```
sqlite> SELECT * FROM COMPANY WHERE AGE NOT IN ( 25, 27 );

ID           NAME        AGE         ADDRESS     SALARY

----------   ----------  ----------  ----------  ----------

1            Paul        32          California  20000.0

3            Teddy       23          Norway      20000.0

6            Kim         22          South-Hall  45000.0

7            James       24          Houston     10000.0
```

Following SELECT statement lists down all the records where AGE value is in BETWEEN 25 AND 27.

```
sqlite> SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement makes use of SQL sub-query where sub-query finds all the records with AGE field having SALARY > 65000 and later WHERE clause is being used along with EXISTS operator to list down all the records where AGE from the outside query exists in the result returned by the sub-query.

```
sqlite> SELECT AGE FROM COMPANY
        WHERE EXISTS (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
AGE
----------
32
25
23
25
27
22
24
```

Following SELECT statement makes use of SQL sub-query where subquery finds all the records with AGE field having SALARY > 65000 and later WHERE clause is being used along with > operator to list down all the records where AGE from the outside query is greater than the age in the result returned by the sub-query.

```
sqlite> SELECT * FROM COMPANY
        WHERE AGE > (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

SQLite Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. Following is the truth table for **&** and **|**.

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, then 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:

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 - Bitwise Operators Example

Following are simple examples showing the usage of SQLite Bitwise Operators.

```
sqlite> .mode line
sqlite> select 60 | 13;
60 | 13 = 61


sqlite> select 60 & 13;
60 & 13 = 12


sqlite> select  60 ^ 13;
10 * 20 = 200



sqlite>  select  (~60);
(~60) = -61


sqlite>  select  (60 << 2);
(60 << 2) = 240


sqlite>  select  (60 >> 2);
(60 >> 2) = 15
```

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 a specific set of data.

Syntax

Consider the basic syntax of the SELECT statement as follows:

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

Following are the different types of SQLite expressions.

SQLite - Boolean Expression

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;
```

Consider COMPANY table with the following records.

```
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
```

Following is a simple example showing the usage of SQLite Boolean Expressions.

```
sqlite> SELECT * FROM COMPANY WHERE SALARY = 10000;
ID          NAME        AGE         ADDRESS     SALARY
```

testDB.db

.   testDB.db

.   ,

.

select age+ 1, salary * 1.1 from company;

.

select age+ 1 as age, salary * 1.1 as salary from company;

```
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 example showing uthe sage of SQLite Numeric Expression.

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

select (15+ 6) as addition from company;

There are several built-in functions such as **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 Expression

Date Expression returns the current system date and time values. These expressions are used in various data manipulations.

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

select current_date;
select current_time;

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 the specific value from the table. You will have to use WHERE clause to filter the records and fetch only the necessary records.

The WHERE clause not only is used in SELECT statement, but it is also used in UPDATE, DELETE statement, etc., which will be covered in subsequent chapters.

Syntax

Following is the basic syntax of SQLite SELECT statement with WHERE clause.

```
SELECT column1, column2, columnN
FROM table_name           (boolean expression)
WHERE [condition]
```

```
select                   : from                                  ,
                 where                                      ,
                         .

      select                        (
  )            , where                                    .
```

```
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
```

Following is a simple example showing the usage of SQLite Logical Operators. Following SELECT statement lists down all the records where AGE is greater than or equal to 25 **AND** salary is greater than or equal to 65000.00.

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE is greater than or equal to 25 **OR** salary is greater than or equal to 65000.00.

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

1           Paul        32          California  20000.0

2           Allen       25          Texas       15000.0

4           Mark        25          Rich-Mond   65000.0

5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE is not NULL, which means all the records because none of the record has AGE equal to NULL.

```
sqlite>  SELECT * FROM COMPANY WHERE AGE IS NOT NULL;
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
```

Following SELECT statement lists down all the records where NAME starts with 'Ki', does not matter what comes after 'Ki'.

```
sqlite> SELECT * FROM COMPANY WHERE NAME LIKE 'Ki%';
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

6           Kim         22          South-Hall  45000.0
```

%, _   .

) '__park%'

Following SELECT statement lists down all the records where NAME starts with 'Ki', does not matter what comes after 'Ki'.

```
sqlite> SELECT * FROM COMPANY WHERE NAME GLOB 'Ki*';
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

6           Kim         22          South-Hall  45000.0
```

*, ?, [   ]

) '?at*Letter[0- 9]'

Following SELECT statement lists down all the records where AGE value is either 25 or 27.

47

```
sqlite> SELECT * FROM COMPANY WHERE AGE IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement lists down all the records where AGE value is neither 25 nor 27.

```
sqlite> SELECT * FROM COMPANY WHERE AGE NOT IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

Following SELECT statement lists down all the records where AGE value is in BETWEEN 25 AND 27.

```
sqlite> SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Following SELECT statement makes use of SQL sub-query, where sub-query finds all the

select   (query)   , select   select
   .   select   (subquery)   (inner query)
        .   (main query)   (outer
query)   .

```
sqlite> SELECT AGE FROM COMPANY
        WHERE EXISTS (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
AGE
----------
32
25
23
25
27
```

exists   , select
        . exists   FALSE,
              TRUE   .
If the subquery returns at least one row, the result is true.
Otherwise, the EXISTS operator returns false or NULL.

,

,
      .

```
22

24
```

Following SELECT statement makes use of SQL sub-query, where sub-query finds all the records with AGE field having SALARY > 65000 and later WHERE clause is being used along with > operator to list down all the records where AGE from the outside query is greater than the age in the result returned by the sub-query.

```
sqlite> SELECT * FROM COMPANY

       WHERE AGE > (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

1           Paul        32          California  20000.0
```

The SQLite **AND** & **OR** operators are used to compile multiple conditions to narrow down the selected data in an SQLite statement. These two operators are called **conjunctive operators**.

These operators provide a means to make multiple comparisons with different operators in the same SQLite statement.

The AND Operator

The **AND** operator allows the existence of multiple conditions in a 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

Following is the basic syntax of AND operator with WHERE clause.

```
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.

Example

Consider COMPANY table with the following records.

```
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
```

Following SELECT statement lists down all the records where AGE is greater than or equal to 25 **AND** salary is greater than or equal to 65000.00.

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

The OR Operator

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

Syntax

Following is the basic syntax of OR operator with WHERE clause.

```
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.

Example

Consider COMPANY table with the following records.

```
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
```

Following SELECT statement lists down all the records where AGE is greater than or equal to 25 **OR** salary is greater than or equal to 65000.00.

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

(   ) scott.db   .

1. scott.db   schema diagram   DB
   .
2.   7521, 7654, 7844 (   )   .
3. emp   mgr   . mgr   null
                        ,   .
4.   1500   ,   ,   .
5.   scott   ,   ,   .
6. emp   hiredate   'yyyy- mm- dd'
        . '1980- 03- 01'   .
7. emp   deptno   job
      . 10   MANAGER   .
8. 10   ,   .
9.   2000   3000   ,   ,   .
10.   .
11. '1987- 01- 01'   '1987- 12- 31'   .

(   ) HR.db   SQL   .

1.   (job)   (job_id)   9   ,   , job_id   .
2. job_id   9   .
3.   Programmer   ,   , job_id   .
4. Jennifer Whalen   .
5. Jennifer Whalen   .
6. Jennifer Whalen   ?
7. Jennifer Whalen   ?
8. 1990   .
9.   .
10. Seattle   ?
11. Seattle   .
12. Seattle   .

DB   insert, update, delete   .
        DB   .
    ,   ,   (log)
                         .
        .   .

Following is the basic syntax of UPDATE query with WHERE clause.

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

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

Example

Consider COMPANY table with the following records.

```
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
```

Following is an example, which will update ADDRESS for a customer whose ID is 6.

```
sqlite> UPDATE COMPANY SET ADDRESS = 'Texas' WHERE ID = 6;
```

Now, COMPANY table will have the following records.

```
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          Texas       45000.0
7          James       24          Houston     10000.0
```

If you want to modify all ADDRESS and SALARY column values in COMPANY table, you do not need to use WHERE clause and UPDATE query will be as follows:

```
sqlite> UPDATE COMPANY SET ADDRESS = 'Texas', SALARY = 20000.00;
```

Now, COMPANY table will have the following records.

```
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

1           Paul        32          Texas       20000.0

2           Allen       25          Texas       20000.0

3           Teddy       23          Texas       20000.0

4           Mark        25          Texas       20000.0

5           David       27          Texas       20000.0

6           Kim         22          Texas       20000.0
```

.   create table ... as select   .

select   SQL   .

.

create table test.company_copy as select * from test.company;

(   ) scott.db   scott   attach
SQL   .

1. emp   dept   emp01, dept01   .
    emp01   dept01   .
2. 10   30   .
3.   3000   10%   .
4.   10000   5000   .
5. scott   20   MANAGER   . (...)
6. scott   ,   50   ,   4000   .
7.   10   .
8.   10%   .
9. emp01   .

(   ) HR.db   SQL   .

1. Kimberly Grant   ?
2. Kimberly Grant   Alexander Hunold   IT   .
SQL   ?

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

Syntax

Following is the basic syntax of DELETE query with WHERE clause.

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

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

Example

Consider COMPANY table with the following records.

```
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
```

Following is an example, which will DELETE a customer whose ID is 7.

```
sqlite> DELETE FROM COMPANY WHERE ID = 7;
```

Now COMPANY table will have the following records.

```
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
```

If you want to DELETE all the records from COMPANY table, you do not need to use WHERE clause with DELETE query, which will be as follows:

```
sqlite> DELETE FROM COMPANY;
```

Now, COMPANY table does not have any record as all the records have been deleted by DELETE statement.

( ) scott.db   scott   attach   SQL
              .

1. emp   emp01   ,   emp01
    .
2. 10   .   . (

          .
    .)
3. emp01   .

( ) chinook.db   mychinook.db   .

1. K- pop   ?
(.)

delete   where
    . delete   where   select
                                    .

SQLite **LIKE** operator is used to match text values against a pattern using wildcards. If the search expression can be matched to the pattern expression, the LIKE operator will return true, which is 1. There are two wildcards used in conjunction with the LIKE operator:

- The percent sign (%) like
- The underscore (_)   .

The percent sign represents zero, one, or multiple numbers or characters. The underscore represents a single number or character. These symbols can be used in combinations.

Syntax

Following is the basic syntax of **%** and **_**.

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'


or


SELECT FROM table_name
WHERE column LIKE '%XXXX%'


or


SELECT FROM table_name
WHERE column LIKE 'XXXX_'


or


SELECT FROM table_name
WHERE column LIKE '_XXXX'


or


SELECT FROM table_name
WHERE column LIKE '_XXXX_'
```

You can combine **N** number of conditions using AND or OR operators. Here, <mark>XXXX could be any numeric or string value.</mark>

like

SQLite

DBMS

Statement	Description
WHERE SALARY LIKE '200%'	Finds any values that start with 200
WHERE SALARY LIKE '%200%'	Finds any values that have 200 in any position
WHERE SALARY LIKE '_00%'	Finds any values that have 00 in the second and third positions
WHERE SALARY LIKE '2_%_%'	Finds any values that start with 2 and are at least 3 characters in length
WHERE SALARY LIKE '%2'	Finds any values that end with 2
WHERE SALARY LIKE '_2%3'	Finds any values that has a 2 in the second position and ends with a 3
WHERE SALARY LIKE '2___3'	Finds any values in a five-digit number that starts with 2 and ends with 3

Let us take a real example, consider COMPANY table with the following records.

```
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23
4           Mark        25
5           David       27
6           Kim         22
7           James       24
```

Following is an example, which will display all the records from COMPANY table where AGE starts with 2.

```
sqlite> SELECT * FROM COMPANY WHERE AGE  LIKE '2%';
```

This will produce the following result.

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
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
```

Following is an example, which will display all the records from COMPANY table where ADDRESS will have a hyphen (-) inside the text.

```
sqlite> SELECT * FROM COMPANY WHERE ADDRESS  LIKE '%-%';
```

This will produce the following result.

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
6           Kim         22          South-Hall  45000.0
```

tutorialspoint
SIMPLYEASYLEARNING

DBMS   glob   .   like
          .   oracle   (regular expression)
  regexp_like(,   )   .
                                            text mining
        .

SQLite **GLOB** operator is used to match only text values against a pattern using wildcards. If the search expression can be matched to the pattern expression, the GLOB operator will return true, which is 1. Unlike LIKE operator, GLOB is case sensitive and it follows syntax of UNIX for specifying THE following wildcards.

sqlite   like   glob   :
glob

                    .

- The asterisk sign (*)
- The question mark (?)

glob

*:
?:
[abcd]: a, b, c, d
[a- t]: a   t
[^abcd]: a, b, c, d,

(   ) select … where filename glob '*[_.,:;!?- (){}< >]OK';

  or

  SELECT FROM table_name
  WHERE column GLOB '*XXXX*'

  or

  SELECT FROM table_name
  WHERE column GLOB 'XXXX?'

  or

  SELECT FROM table_name
  WHERE column GLOB '?XXXX'

  or

  SELECT FROM table_name
  WHERE column GLOB '?XXXX?'

  or

```
SELECT FROM table_name
WHERE column GLOB '????'
```

You can combine **N** number of conditions using AND or OR operators. Here, XXXX could be any numeric or string value.

Example

Following table lists a number of examples showing WHERE part having different LIKE clause with '*****' and '**?**' operators.

Statement	Description
WHERE SALARY GLOB '200*'	Finds any values that start with 200
WHERE SALARY GLOB '*200*'	Finds any values that have 200 in any position
WHERE SALARY GLOB '?00*'	Finds any values that have 00 in the second and third positions
WHERE SALARY GLOB '2??'	Finds any values that start with 2 and are at least 3 characters in length
WHERE SALARY GLOB '*2'	Finds any values that end with 2
WHERE SALARY GLOB '?2*3'	Finds any values that have a 2 in the second position and end with a 3
WHERE SALARY GLOB '2???3'	Finds any values in a five-digit number that start with 2 and end with 3

Let us take a real example, consider COMPANY table with the following records.

```
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
```

Following is an example, which will display all the records from COMPANY table, where AGE starts with 2.

```
sqlite> SELECT * FROM COMPANY WHERE AGE  GLOB '2*';
```

This will produce the following result.

```
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

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
```

Following is an example, which will display all the records from COMPANY table where ADDRESS will have a hyphen (-) inside the text.

```
sqlite> SELECT * FROM COMPANY WHERE ADDRESS  GLOB '*-*';
```

This will produce the following result.

```
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

4           Mark        25          Rich-Mond   65000.0

6           Kim         22          South-Hall  45000.0
```

(   ) scott.db   SQL   .

1.   J   .
2. 1   csv   EXCEL   .
3.   K   .
4.   K   .
5.   A   .

(   ) chinookdb   SQL   .

1.   .
2.   (media type)   MPEG- 4   id   .
3.   Man   .
4.   Man   .
5.   2- 4   ere   .
6.   .
7.   .

limit   select   .

Syntax

Following is the basic syntax of SELECT statement with LIMIT clause.

```
SELECT column1, column2, columnN

FROM table_name

LIMIT [no of rows]
```

Following is the syntax of LIMIT clause when it is used along with OFFSET clause.

```
SELECT column1, column2, columnN

FROM table_name

LIMIT [no of rows] OFFSET [row num]
```

SQLite engine will return rows starting from the next row to the given OFFSET as shown below in the last example.

Example

Consider COMPANY table with the following records.

```
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
```

Following is an example, which limits the row in the table according to the number of rows you want to fetch from table.

```
sqlite> SELECT * FROM COMPANY LIMIT 6;
```

This will produce the following result.

```
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
```

However in certain situations, you may need to pick up a set of records from a particular offset. Here is an example, which picks up 3 records starting from the 3rd position.

```
sqlite> SELECT * FROM COMPANY LIMIT 3 OFFSET 2;
```

This will produce the following result.

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

order by   .
                                        . select   .

SQLite **ORDER BY** clause is used to sort the data in an ascending or descending order, based on one or more columns.

Syntax

Following is the basic syntax of ORDER BY clause.

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

select
                    .

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be available in the column-list.

Example

Consider COMPANY table with the following records.

```
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
```

Following is an example, which will sort the result in descending order by SALARY.

```
sqlite> SELECT * FROM COMPANY ORDER BY SALARY ASC;
```

This will produce the following result.

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

Following is an example, which will sort the result in descending order by NAME and SALARY.

```
sqlite> SELECT * FROM COMPANY ORDER BY NAME, SALARY ASC;
```

This will produce the following result.

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

Following is an example, which will sort the result in descending order by NAME.

```
sqlite> SELECT * FROM COMPANY ORDER BY NAME DESC;
```

This will produce the following result.

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

(   ) chinook.db   SQL   .

1.   ,   ,
      .

2.   .

GROUP BY

.   select   ,
.

        (aggregate function)   sum(), avg(), count(), max(), min(),
group_concat()   ,   .
group_concat()

        .

group by   .
group by   where   , order by   .

```
SELECT column-list

FROM table_name

WHERE [ conditions ]

GROUP BY column1, column2....columnN

ORDER BY column1, column2....columnN
```

You can use more than one column in the GROUP BY clause. Make sure whatever column you are using to group, that column should be available in the column-list.

Example

Consider COMPANY table with the following records.

```
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
```

If you want to know the total amount of salary on each customer, then GROUP BY query will be as follows:

```
sqlite> SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME;
```

This will produce following result.

```
NAME        SUM(SALARY)
----------  -----------
Allen       15000.0
David       85000.0
James       10000.0
Kim         45000.0
Mark        65000.0
Paul        20000.0
Teddy       20000.0
```

company

Now, let us create three more records in COMPANY table using the following INSERT statements.

```
INSERT INTO COMPANY VALUES (8, 'Paul', 24, 'Houston', 20000.00 );

INSERT INTO COMPANY VALUES (9, 'James', 44, 'Norway', 5000.00 );

INSERT INTO COMPANY VALUES (10, 'James', 45, 'Texas', 5000.00 );
```

Now, our table has the following records with duplicate names.

```
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
8           Paul        24          Houston     20000.0
9           James       44          Norway      5000.0
10          James       45          Texas       5000.0
```

Again, let us use the same statement to group-by all the records using NAME column as follows:

```
sqlite> SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME ORDER BY NAME;
```

This will produce the following result.

```
NAME        SUM(SALARY)
----------  -----------
Allen       15000
David       85000
James       20000
Kim         45000
Mark        65000
Paul        40000
Teddy       20000
```

Let us use ORDER BY clause along with GROUP BY clause as follows:

```
sqlite>  SELECT NAME, SUM(SALARY)
         FROM COMPANY GROUP BY NAME ORDER BY NAME DESC;
```

This will produce the following result.

```
NAME        SUM(SALARY)
----------  -----------
Teddy       20000
Paul        40000
Mark        65000
Kim         45000
James       20000
David       85000
Allen       15000
```

(   ) scott.db   select   .

1.   .
2.   .
3.   (job)   ?
4.   ?
5.   ?
6.   ?
7.   ,   ,   .
8.
  . (?)

(   ) chinook.db

1.   track   ,   track   ,   ,
                              .

(   ) northwind.sql   northwind.db
    . ($ sqlite3 northwind.db < northwind.sql)

1.   ,   ,   ,

having   group by   . group by
                                                    having   .
                                      .   having   group by
      . order by   order by   .

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by GROUP BY clause.

Syntax

Following is the position of HAVING clause in a SELECT query.

```
SELECT

FROM

WHERE

GROUP BY

HAVING

ORDER BY
```

HAVING clause must follow GROUP BY clause in a query and must also precede ORDER BY clause, if used. Following is the syntax of the SELECT statement, including HAVING clause.

```
SELECT column1, column2

FROM table1, table2

WHERE [ conditions ]

GROUP BY column1, column2

HAVING [ conditions ]

ORDER BY column1, column2
```

from   2   .
      join   .
      .

Example

Consider COMPANY table with the following records.

```
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
```

70

8	Paul	24	Houston	20000.0
9	James	44	Norway	5000.0
10	James	45	Texas	5000.0

Following is the example, which will display the record for which the name count is less than 2.

```
sqlite > SELECT * FROM COMPANY GROUP BY name HAVING count(name) < 2;
```

This will produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
2	Allen	25	Texas	15000
5	David	27	Texas	85000
6	Kim	22	South-Hall	45000
4	Mark	25	Rich-Mond	65000
3	Teddy	23	Norway	20000

Following is the example, which will display the record for which the name count is greater than 2.

```
sqlite > SELECT * FROM COMPANY GROUP BY name HAVING count(name) > 2;
```

This will produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
10	James	45	Texas	5000

(   ) scott.db   .

1.   2000   .
7.   2900   .

(   ) chinook.db

1.   5   .
2. 100   track   track   .

SQLite **DISTINCT** keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetch only the unique records.

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

Syntax

Following is the basic syntax of DISTINCT keyword to eliminate duplicate records.

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

Example

Consider COMPANY table with the following records.

```
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
8           Paul        24          Houston     20000.0
9           James       44          Norway      5000.0
10          James       45          Texas       5000.0
```

First, let us see how the following SELECT query returns duplicate salary records.

```
sqlite> SELECT name FROM COMPANY;
```

This will produce the following result.

```
NAME
----------
Paul

Allen

Teddy

Mark

David

Kim

James

Paul

James

James
```

Now, let us use **DISTINCT** keyword with the above SELECT query and see the result.

```
sqlite> SELECT DISTINCT name FROM COMPANY;
```

This will produce the following result, where there is no duplicate entry.

```
NAME
----------
Paul

Allen

Teddy

Mark

David

Kim

James
```

(   ) scott.db

1.   .   ?
2.   ?   ?

(   ) northwind.db

1.   .
2.   .

tutorialspoint
SIMPLYEASYLEARNING

Advanced SQLite

SQLite **PRAGMA** command is a special command to be used to control various environmental variables and state flags within the SQLite environment. A PRAGMA value can be read and it can also be set based on the requirements.

Syntax

To query the current PRAGMA value, just provide the name of the pragma.

```
PRAGMA pragma_name;
```

To set a new value for PRAGMA, use the following syntax.

```
PRAGMA pragma_name = value;
```

The set mode can be either the name or the integer equivalent but the returned value will always be an integer.

auto_vacuum Pragma

The **auto_vacuum** pragma gets or sets the auto-vacuum mode. Following is the simple syntax.

```
PRAGMA [database.]auto_vacuum;
PRAGMA [database.]auto_vacuum = mode;
```

Where **mode** can be any of the following:

Pragma Value	Description
0 or NONE	Auto-vacuum is disabled. This is the default mode which means that a database file will never shrink in size unless it is manually vacuumed using the VACUUM command.
1 or FULL	Auto-vacuum is enabled and fully automatic which allows a database file to shrink as data is removed from the database.
2 or INCREMENTAL	Auto-vacuum is enabled but must be manually activated. In this mode, the reference data is maintained, but free pages are simply put on the free list. These pages can be recovered using the **incremental_vacuum pragma** any time.

cache_size Pragma

The **cache_size** pragma can get or temporarily set the maximum size of the in-memory page cache. Following is the simple syntax.

```
PRAGMA [database.]cache_size;

PRAGMA [database.]cache_size = pages;
```

The **pages** value represents the number of pages in the cache. The built-in page cache has a default size of 2,000 pages and a minimum size of 10 pages.

case_sensitive_like Pragma

The **case_sensitive_like** pragma controls the case-sensitivity of the built-in LIKE expression. By default, this pragma is false which means that the built-in LIKE operator ignores the letter case. Following is the simple syntax.

```
PRAGMA case_sensitive_like = [true|false];
```

There is no way to query for the current state of this pragma.

count_changes Pragma

count_changes pragma gets or sets the return value of data manipulation statements such as INSERT, UPDATE and DELETE. Following is the simple syntax.

```
PRAGMA count_changes;

PRAGMA count_changes = [true|false];
```

By default, this pragma is false and these statements do not return anything. If set to true, each of the mentioned statement will return a one-column, one-row table consisting of a single integer value, indicating impacted rows by the operation.

database_list Pragma

The **database_list** pragma will be used to list down all the databases attached. Following is the simple syntax.

```
PRAGMA database_list;
```

This pragma will return a three-column table with one row per open or attached database giving database sequence number, its name and the file associated.

encoding Pragma

The **encoding** pragma controls how strings are encoded and stored in a database file. Following is the simple syntax.

```
PRAGMA encoding;
```

```
PRAGMA encoding = format;
```

The format value can be one of **UTF-8**, **UTF-16le**, or **UTF-16be**.

freelist_count Pragma

The **freelist_count** pragma returns a single integer indicating how many database pages are currently marked as free and available. Following is the simple syntax.

```
PRAGMA [database.]freelist_count;
```

The format value can be one of **UTF-8**, **UTF-16le**, or **UTF-16be**.

index_info Pragma

The **index_info** pragma returns information about a database index. Following is the simple syntax.

```
PRAGMA [database.]index_info( index_name );
```

The result set will contain one row for each column contained in the index giving column sequence, column index within table and column name.

index_list Pragma

index_list pragma lists all of the indexes associated with a table. Following is the simple syntax.

```
PRAGMA [database.]index_list( table_name );
```

The result set will contain one row for each index giving index sequence, index name and flag indicating whether the index is unique or not.

journal_mode Pragma

The **journal_mode** pragma gets or sets the journal mode which controls how the journal file is stored and processed. Following is the simple syntax.

```
PRAGMA journal_mode;
PRAGMA journal_mode = mode;
PRAGMA database.journal_mode;
PRAGMA database.journal_mode = mode;
```

There are five supported journal modes as listed in the following table.

Pragma Value	Description
DELETE	This is the default mode. Here at the conclusion of a transaction, the journal file is deleted.

TRUNCATE	The journal file is truncated to a length of zero bytes.
PERSIST	The journal file is left in place, but the header is overwritten to indicate the journal is no longer valid.
MEMORY	The journal record is held in memory, rather than on disk.
OFF	No journal record is kept.

max_page_count Pragma

The **max_page_count** pragma gets or sets the maximum allowed page count for a database. Following is the simple syntax.

```
PRAGMA [database.]max_page_count;
PRAGMA [database.]max_page_count = max_page;
```

The default value is 1,073,741,823 which is one giga-page, which means if the default is 1 KB page size, this allows databases to grow up to one terabyte.

page_count Pragma

The **page_count** pragma returns the current number of pages in the database. Following is the simple syntax.

```
PRAGMA [database.]page_count;
```

The size of the database file should be page_count * page_size.

page_size Pragma

The **page_size** pragma gets or sets the size of the database pages. Following is the simple syntax.

```
PRAGMA [database.]page_size;
PRAGMA [database.]page_size = bytes;
```

By default, the allowed sizes are 512, 1024, 2048, 4096, 8192, 16384, and 32768 bytes. The only way to alter the page size on an existing database is to set the page size and then immediately VACUUM the database.

parser_trace Pragma

The **parser_trace** pragma controls printing the debugging state as it parses SQL commands. Following is the simple syntax.

```
PRAGMA parser_trace = [true|false];
```

By default, it is set to false but when enabled by setting it to true, the SQL parser will print its state as it parses SQL commands.

recursive_triggers Pragma

The **recursive_triggers** pragma gets or sets the recursive trigger functionality. If recursive triggers are not enabled, a trigger action will not fire another trigger. Following is the simple syntax.

```
PRAGMA recursive_triggers;
PRAGMA recursive_triggers = [true|false];
```

schema_version Pragma

The **schema_version** pragma gets or sets the schema version value that is stored in the database header. Following is the simple syntax.

```
PRAGMA [database.]schema_version;
PRAGMA [database.]schema_version = number;
```

This is a 32-bit signed integer value that keeps track of schema changes. Whenever a schema-altering command is executed (like, CREATE... or DROP...), this value is incremented.

secure_delete Pragma

The **secure_delete** pragma is used to control how the content is deleted from the database. Following is the simple syntax.

```
PRAGMA secure_delete;
PRAGMA secure_delete = [true|false];
PRAGMA database.secure_delete;
PRAGMA database.secure_delete = [true|false];
```

The default value for the secure delete flag is normally off, but this can be changed with the SQLITE_SECURE_DELETE build option.

sql_trace Pragma

The **sql_trace** pragma is used to dump SQL trace results to the screen. Following is the simple syntax.

```
PRAGMA sql_trace;
PRAGMA sql_trace = [true|false];
```

SQLite must be compiled with the SQLITE_DEBUG directive for this pragma to be included.

synchronous Pragma

The **synchronous** pragma gets or sets the current disk synchronization mode, which controls how aggressively SQLite will write data all the way out to physical storage. Following is the simple syntax.

```
PRAGMA [database.]synchronous;

PRAGMA [database.]synchronous = mode;
```

SQLite supports the following synchronization modes as listed in the table.

Pragma Value	Description
0 or OFF	No syncs at all
1 or NORMAL	Sync after each sequence of critical disk operations
2 or FULL	Sync after each critical disk operation

temp_store Pragma

The **temp_store** pragma gets or sets the storage mode used by temporary database files. Following is the simple syntax.

```
PRAGMA temp_store;

PRAGMA temp_store = mode;
```

SQLite supports the following storage modes.

Pragma Value	Description
0 or DEFAULT	Use compile-time default. Normally FILE.
1 or FILE	Use file-based storage.
2 or MEMORY	Use memory-based storage.

temp_store_directory Pragma

The **temp_store_directory** pragma gets or sets the location used for temporary database files. Following is the simple syntax.

```
PRAGMA temp_store_directory;

PRAGMA temp_store_directory = 'directory_path';
```

user_version Pragma

The **user_version** pragma gets or sets the user-defined version value that is stored in the database header. Following is the simple syntax.

```
PRAGMA [database.]user_version;

PRAGMA [database.]user_version = number;
```

This is a 32-bit signed integer value, which can be set by the developer for version tracking purpose.

writable_schema Pragma

The **writable_schema** pragma gets or sets the ability to modify system tables. Following is the simple syntax.

```
PRAGMA writable_schema;

PRAGMA writable_schema = [true|false];
```

If this pragma is set,tables that start with sqlite_ can be created and modified, including the sqlite_master table. Be careful while using pragma because it can lead to complete database corruption.

to one column, whereas table level constraints are applied to the whole table.

Following are commonly used constraints available in SQLite.

- **NOT NULL Constraint**: Ensures that a column cannot have NULL value.

- **DEFAULT Constraint**: Provides a default value for a column when none is specified.

  Foreign Key constraint???

- **UNIQUE Constraint**: Ensures that all values in a column are different.

- **PRIMARY Key**: Uniquely identifies each row/record in a database table.

- **CHECK Constraint**: Ensures that all values in a column satisfies certain conditions.

NOT NULL Constraint

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column.

A NULL is not the same as no data, rather, it represents unknown data.

Example

For example, the following SQLite statement creates a new table called COMPANY and adds five columns, three of which, ID and NAME and AGE, specifies not to accept NULLs.

```
CREATE TABLE COMPANY(                  not null
    ID INT PRIMARY KEY     NOT NULL,          . (SQLite)
    NAME           TEXT    NOT NULL,
    AGE            INT     NOT NULL,
    ADDRESS        CHAR(50),
    SALARY         REAL
);
```

DEFAULT Constraint

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

Example

For example, the following SQLite statement creates a new table called COMPANY and adds five columns. Here, SALARY column is set to 5000.00 by default, thus in case INSERT INTO statement does not provide a value for this column, then by default, this column would be set to 5000.00.

```
CREATE TABLE COMPANY(
    ID INT PRIMARY KEY     NOT NULL,
    NAME            TEXT    NOT NULL,          default
    AGE             INT     NOT NULL,                    . (SQLite)
    ADDRESS         CHAR(50),
    SALARY          REAL    DEFAULT 50000.00
);
```

UNIQUE Constraint

The UNIQUE Constraint prevents two records from having identical values in a particular column. In the COMPANY table, for example, you might want to prevent two or more people from having an identical age.

Example

unique   null   . null
                                                            .
For example, the fo primary key   not null + unique
adds five columns.
                            .
records with the same age.

```
CREATE TABLE COMPANY(
    ID INT PRIMARY KEY     NOT NULL,
    NAME            TEXT    NOT NULL,
    AGE             INT     NOT NULL UNIQUE,
    ADDRESS         CHAR(50),
    SALARY          REAL    DEFAULT 50000.00
);
```

unique   ,   .

constraint age_uk unique(age)

PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table. There can be more UNIQUE columns, but only one primary key in a table. Primary keys are important when designing the database tables. Primary keys are unique IDs.

We use them to refer to table rows. Primary keys become foreign keys in other tables, when creating relations among tables. Due to a 'longstanding coding oversight', primary keys can be NULL in SQLite. This is not the case with other databases.

A primary key is a field in a table which uniquely identifies each rows/records in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a **composite key**.

sqlite   primary key   not null
    .   DBMS   .   primary key
          unique + not null   .

Example

You already have seen various examples above where we have created COMPANY table with ID as a primary key.

```
CREATE TABLE COMPANY(

   ID INT  PRIMARY KEY     NOT NULL,

   NAME            TEXT    NOT NULL,

   AGE             INT     NOT NULL,

   ADDRESS         CHAR(50),

   SALARY          REAL

);
```

primary key
                .

                .

constraint company_pk primary key(id)

CHECK Constraint

CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn't entered into the table.

Example

For example, the following SQLite creates a new table called COMPANY and adds five columns. Here, we add a CHECK with SALARY column, so that you cannot have any SALARY Zero.

```
CREATE TABLE COMPANY3(

   ID INT PRIMARY KEY     NOT NULL,

   NAME            TEXT    NOT NULL,

   AGE             INT     NOT NULL,

   ADDRESS         CHAR(50),
```

84

```
    SALARY            REAL      CHECK(SALARY > 0)

);
```

Dropping Constraint

SQLite supports a limited subset of ALTER TABLE. The ALTER TABLE command in SQLite allows the user to rename a table or add a new column to an existing table. It is not possible to rename a column, remove a column, or add or remove constraints from a table.

DBMS   alter table   .
SQLite   alter table   .

alter table
    add constraint   ;

alter table
    drop constraint   ;

foreign key   . foreign key
                                    .
                      .   foreign key
            . foreign key   primary
key   (reference)   .

constraint tbla_tblb_fk foreign key (_   ) references   _
            (_   _   )

(   ) northwind DB   schema diagram
                  .

85

SQLite **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

SQL defines three major types of joins –

- The CROSS JOIN
- The INNER JOIN
- The OUTER JOIN

schema diagram

.

Before we proceed, let's consider two tables COMPANY and DEPARTMENT. We already have seen INSERT statements to populate COMPANY table. So just let's assume the list of records available in COMPANY table –

```
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
```

Another table is DEPARTMENT with the following definition –

```
CREATE TABLE DEPARTMENT(
    ID INT PRIMARY KEY      NOT NULL,
    DEPT            CHAR(50) NOT NULL,
    EMP_ID          INT      NOT NULL
);
```

Here is the list of INSERT statements to populate DEPARTMENT table –

```
INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (1, 'IT Billing', 1 );


INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (2, 'Engineering', 2 );
```

```
INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)

VALUES (3, 'Finance', 7 );
```

Finally, we have the following list of records available in DEPARTMENT table −

```
ID          DEPT        EMP_ID

----------  ----------  ----------

1           IT Billing  1

2           Engineering 2

3           Finance     7
```

The CROSS JOIN

CROSS JOIN matches eve   cross join   .
the input tables have **x**   .   Cartesian product   . cross join
columns. Because CROSS
care must be taken to only use them when appropriate.

Following is the syntax of CROSS JOIN −

```
SELECT ... FROM table1 CROSS JOIN table2 ...
```

Based on the above tables, you can write a CROSS JOIN as follows −

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY CROSS JOIN DEPARTMENT;
```

The above query will produ

```
EMP_ID      NAME

----------  ----------

1           Paul

2           Paul        Engineering

7           Paul        Finance

1           Allen       IT Billing

2           Allen       Engineering

7           Allen       Finance

1           Teddy       IT Billing

2           Teddy       Engineering

7           Teddy       Finance

1           Mark        IT Billing

2           Mark        Engineering

7           Mark        Finance

1           David       IT Billing
```

87

2	David	Engineering
7	David	Finance
1	Kim	IT Billing
2	Kim	Engineering
7	Kim	Finance
1	James	IT Billing
2	James	Engineering
7	James	Finance

The INNER JOIN

cross join

inner join   .

inner join   .

INNER JOIN creates a n
and table2) based upor
each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, the column values for each matched pair of rows of A and B are combined into a result row.

An INNER JOIN is the most common and default type of join. You can use INNER keyword optionally.

table1.   = table2.
        table1.   = table2.

Following is the syntax of INNER JOIN −

```
SELECT ... FROM table1 [INNER] JOIN table2 ON conditional_expression ...
```

To avoid redundancy and keep the phrasing shorter,
declared with a **USING** expression. This expression specif

```
SELECT ... FROM table1 JOIN table2 USING ( column1 ,... ) ...
```

A NATURAL JOIN is similar to a **JOIN…USING**, only it automatically tests for equality between the values of every column that exists in both tables −

```
SELECT ... FROM table1 NATURAL JOIN table2...
```

Based on the above tables, you can write an INNER JOIN as follows −

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
       ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

The above query will produce the following result −

EMP_ID	NAME	DEPT
1	Paul	IT Billing
2	Allen	Engineering
7	James	Finance

join

inner join

(natural join)   .

. (...)

The OUTER JOIN

OUTER JOIN is an exten...
OUTER JOINs: LEFT, RIG...

OUTER JOINs have a co...
USING, or NATURAL key...
the primary JOIN is calc...
both tables, pad them ou...

Following is the syntax o...

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 ON conditional_expression ...
```

To avoid redundancy and keep the phrasing shorter, OUTER JOIN conditions can be declared with a USING expression. This expression specifies a list of one or more columns.

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 USING ( column1 ,... ) ...
```

Based on the above tables, you can write an inner join as follows −

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

The above query will produce the following result −

EMP_ID	NAME	DEPT
----------	----------	----------
1	Paul	IT Billing
2	Allen	Engineerin

inner join
"
"
.
.
outer join .
outer join   full outer join, left outer join, right outer join
.
,
.   full outer join

```
                    from
                      .
SELECT c.EMP_ID, c.NAME, d.DEPT
FROM COMPANY as c LEFT OUTER JOIN DEPARTMENT as d
ON c.ID = d.EMP_ID;
```

(   ) scott.db   .

1.   ,   ,   .
2.   'SCOTT'   ?
3.   .
4. ACCOUNTING   .
6. (self join)   .
7. (self join)   KING   .
8. (self join)   .

                              .

9.   .

              .

(   ) chinook.db   .

1.   "Movies"   playlist   track   ,   .
2. 5   ,   .
3.   track   id,   ,   .
4.   50   (invoice)
              .   ,   ,   ,   (track id, track
    ,   )   .

(   ) northwind.db

    .   ,   .

SQL , sqlite . select
" " , select
, (union), (intersect),
.(except) . select
select (
,) .

Syntax

Following is the basic syntax of **UNION**.

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]


UNION


SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

select . ,
.

intersect, except ,
.

Here the given condition could be any given expression based on your requirement.

Example

Consider the following two tables, (a) COMPANY table as follows:

```
sqlite> select * from COMPANY;
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
```

(b) Another table is <u>DEPARTMENT</u> as follows:

```
ID          DEPT                 EMP_ID
----------  -------------------  ----------
1           IT Billing           1
2           Engineering          2
3           Finance              7
4           Engineering          3
5           Finance              4
6           Engineering          5
7           Finance              6
```

Now let us join these two tables using SELECT statement along with UNION clause as follows:

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID
   UNION
     SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

This will produce the following result:

```
EMP_ID      NAME                 DEPT
----------  -------------------  ----------
1           Paul                 IT Billing
2           Allen                Engineering
3           Teddy                Engineering
4           Mark                 Finance
5           David                Engineering
6           Kim                  Finance
7           James                Finance
```

The UNION ALL Clause

The UNION ALL operator is used to combine the results of two SELECT statements <u>including duplicate rows</u>.

The same rules that apply to UNION apply to the UNION ALL operator as well.

Syntax

Following is the basic syntax of **UNION ALL**.

```
SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]


UNION ALL                        .                                 ...


SELECT column1 [, column2 ]

FROM table1 [, table2 ]

[WHERE condition]
```

Here the given condition could be any given expression based on your requirement.

Example

Now, let us join the above-mentioned two tables in our SELECT statement as follows:

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID
    UNION ALL
      SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

This will produce the following result:

```
EMP_ID      NAME                 DEPT
----------  -------------------- ----------
1           Paul                 IT Billing
2           Allen                Engineering
3           Teddy                Engineering
4           Mark                 Finance
5           David                Engineering
6           Kim                  Finance
7           James                Finance
1           Paul                 IT Billing
```

2	Allen	Engineering
3	Teddy	Engineering
4	Mark	Finance
5	David	Engineering
6	Kim	Finance
7	James	Finance

(   ) 1. scott.db   .

(   ) chinook.db   .

1.   track   id   .
2.   id   .
3. playlist   track   id   .
4.   .

SQLite **NULL** is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.

A field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces.

Syntax

Following is the basic syntax of using **NULL** while creating a table.

```
SQLite> CREATE TABLE COMPANY(
    ID INT PRIMARY KEY     NOT NULL,
    NAME           TEXT    NOT NULL,
    AGE            INT     NOT NULL,
    ADDRESS        CHAR(50),
    SALARY         REAL
);
```

Here, **NOT NULL** signifies that the column should always accept an explicit value of the given data type. There are two columns where we did not use NOT NULL which means these columns could be NULL.

A field with a NULL value is one that has been left blank during record creation.

Example

The NULL value can cause problems when selecting data, because when comparing an unknown value to any other value, the result is always unknown and not included in the final results. Consider the following table, COMPANY with the following records:

```
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
```

Let us use UPDATE statement to set a few nullable values as NULL as follows:

```
sqlite> UPDATE COMPANY SET ADDRESS = NULL, SALARY = NULL where ID IN(6,7);
```

Now, COMPANY table will have the following records.

```
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

7           James       24
```

Next, let us see the usage of **IS NOT NULL** operator to list down all the records where SALARY is not NULL.

```
sqlite> SELECT  ID, NAME, AGE, ADDRESS, SALARY
        FROM COMPANY
        WHERE SALARY IS NOT NULL;
```

The above SQLite statement will produce the following result.

```
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
```

Following is the usage of **IS NULL** operator, which will list down all the records where SALARY is NULL.

```
sqlite> SELECT  ID, NAME, AGE, ADDRESS, SALARY
        FROM COMPANY
        WHERE SALARY IS NULL;
```

The above SQLite statement will produce the following result.

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22
7           James       24
```

You can rename a table or a column temporarily by giving another name, which is known as **ALIAS**. The use of table aliases means to rename a table in a particular SQLite statement. Renaming is a temporary change and the actual table name does not change in the database.

The column aliases are used to rename a table's columns for the purpose of a particular SQLite query.

Syntax

Following is the basic syntax of **table** alias.

```
SELECT column1, column2....

FROM table_name AS alias_name

WHERE [condition];
```

Following is the basic syntax of **column** alias.

```
SELECT column_name AS alias_name

FROM table_name

WHERE [condition];
```

Example

Consider the following two tables, (a) COMPANY table is as follows:

```
sqlite> select * from COMPANY;
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
```

(b) Another table is <u>DEPARTMENT</u> as follows:

```
ID          DEPT                 EMP_ID

----------  --------------------  ----------

1           IT Billing           1

2           Engineering          2

3           Finance              7

4           Engineering          3

5           Finance              4

6           Engineering          5

7           Finance              6
```

Now, following is the usage of **TABLE ALIAS** where we use C and D as aliases for COMPANY and DEPARTMENT tables respectively:

```
sqlite> SELECT C.ID, C.NAME, C.AGE, D.DEPT
        FROM COMPANY AS C, DEPARTMENT AS D
        WHERE  C.ID = D.EMP_ID;
```

select c.id, c.name, c.age, d.dept
from company as c inner join
department as d on (c.id = d.emp_id)

The above SQLite statement will produce the following result.

```
ID          NAME        AGE         DEPT

----------  ----------  ----------  ----------

1           Paul        32          IT Billing

2           Allen       25          Engineerin

3           Teddy       23          Engineerin

4           Mark        25          Finance

5           David       27          Engineerin

6           Kim         22          Finance

7           James       24          Finance
```

Consider an example for the usage of **COLUMN ALIAS** where COMPANY_ID is an alias of ID column and COMPANY_NAME is an alias of name column.

```
sqlite> SELECT C.ID AS COMPANY_ID, C.NAME AS COMPANY_NAME, C.AGE, D.DEPT
        FROM COMPANY AS C, DEPARTMENT AS D
        WHERE  C.ID = D.EMP_ID;
```

select c.id as company_id
, c.name as company_name
, c.age
, d.dept
from company as c inner join department as d on c.id = d.emp_id;

The above SQLite statement will produce the following result.

```
COMPANY_ID  COMPANY_NAME  AGE         DEPT
----------  ------------  ----------  ----------
1           Paul          32          IT Billing
2           Allen         25          Engineerin
3           Teddy         23          Engineerin
4           Mark          25          Finance
5           David         27          Engineerin
6           Kim           22          Finance
7           James         24          Finance
```

SQLite **Triggers** are database callback functions, which are automatically performed/invoked when a specified database event occurs. Following are the important points about SQLite triggers:

- SQLite trigger may be specified to fire whenever a DELETE, INSERT or UPDATE of a particular database table occurs or whenever an UPDATE occurs on one or more specified columns of a table.

- At this time, SQLite supports only FOR EACH ROW triggers, not FOR EACH STATEMENT triggers. Hence, explicitly specifying FOR EACH ROW is optional.

- Both the WHEN clause and the trigger actions may access elements of the row being inserted, deleted, or updated using references of the **form NEW.column-name** and **OLD.column-name**, where column-name is the name of a column from the table that the trigger is associated with.

- If a WHEN clause is supplied, the SQL statements specified are only executed for rows for which the WHEN clause is true. If no WHEN clause is supplied, the SQL statements are executed for all rows.

- The BEFORE or AFTER keyword determines when the trigger actions will be executed relative to the insertion, modification, or removal of the associated row.

- Triggers are automatically dropped when the table that they are associated with is dropped.

- The table to be modified must exist in the same database as the table or view to which the trigger is attached and one must use just **tablename** not **database.tablename**.

- A special SQL function RAISE() may be used within a trigger-program to raise an exception.

Syntax

Following is the basic syntax of creating a **trigger**.

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] event_name

ON table_name  for each row

BEGIN

 -- Trigger logic goes here....

END;
```

Here, **event_name** could be *INSERT, DELETE,* and *UPDATE* database operation on the mentioned table **table_name**. You can optionally specify FOR EACH ROW after table name.

Following is the syntax for creating a trigger on an UPDATE operation on one or more specified columns of a table.

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] UPDATE OF column_name

ON table_name

BEGIN

 -- Trigger logic goes here....

END;
```

Example

Let us consider a case where we want to keep audit trial for every record being inserted in COMPANY table, which we create newly as follows (Drop COMPANY table if you already have it).

```
sqlite> CREATE TABLE COMPANY(

   ID INT PRIMARY KEY     NOT NULL,

   NAME           TEXT    NOT NULL,

   AGE            INT     NOT NULL,

   ADDRESS        CHAR(50),

   SALARY         REAL

);
```

To keep audit trial, we will create a new table called AUDIT where the log messages will be inserted, whenever there is an entry in COMPANY table for a new record.

```
sqlite> CREATE TABLE AUDIT( id integer primary key autoincrement,

    EMP_ID INT NOT NULL,

    ENTRY_DATE TEXT NOT NULL

);
```

Here, ID is the AUDIT record ID, and EMP_ID is the ID which will come from COMPANY table and DATE will keep timestamp when the record will be created in COMPANY table. Now let's create a trigger on COMPANY table as follows:

```
sqlite> CREATE TRIGGER audit_log AFTER INSERT

ON COMPANY                                  current_timestamp

BEGIN

   INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID, datetime('now'));

END;
```

dbvisualizer   trigger
  . SQL   ;   ;
    .   ;   - - / /
                 .

Now, we will start actual work. Let's start inserting record in COMPANY table which should result in creating an audit log record in AUDIT table. Create one record in COMPANY table as follows:

```
sqlite> INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (1, 'Paul', 32, 'California', 20000.00 );
```

This will create one record in COMPANY table, which is as follows:

```
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

1           Paul        32          California  20000.0
```

Same time, one record will be created in AUDIT table. This record is the result of a trigger, which we have created on INSERT operation in COMPANY table. Similarly, you can create your triggers on UPDATE and DELETE operations based on your requirements.

```
EMP_ID      ENTRY_DATE

----------  -------------------

1           2013-04-05 06:26:00
```

Listing Triggers

dbvisualizer free version   fuction, procedure, trigger
      .   .

select * from sql_master where type= 'trigger';

The above SQLite statement will list down only one entry as follows:

```
name

----------

audit_log
```

If you want to list down triggers on a particular table, then use AND clause with table name as follows:

```
sqlite> SELECT name FROM sqlite_master

WHERE type = 'trigger' AND tbl_name = 'COMPANY';
```

The above SQLite statement will also list down only one entry as follows:

```
name

----------

audit_log
```

Dropping Triggers

Following is the DROP command, which can be used to drop an existing trigger.

```
sqlite> DROP TRIGGER trigger_name;
```

(   ) scott.db   emp
                   log   .
emp   not null   .
action   'insert', 'update', 'delete'   .

id, action, old_empid, old_ename, old_mgr, old_sal, old_comm, old_deptno,
new_ename, new_mgr, new_sal, new_comm, new_deptno

1. insert, update, delete   log
trigger   .

index

with UPDATE and INSERT statements. Indexes can be created or dropped with no effect

the index is in an ascending or descending order.

Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

The CREATE INDEX Command

Following is the basic syntax of **CREATE INDEX**.

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes

A single-column index is one that is created based on only one table column. The basic syntax is as follows:

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

Unique Indexes

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows:

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

unique
index

Composite Indexes

A composite index is an index on two or more columns of a table. The basic syntax is as follows:

```
CREATE INDEX index_name

on table_name (column1, column2);
```

where

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

Example

Following is an example where we will create an index in COMPANY table for salary column:

```
sqlite> CREATE INDEX salary_index ON COMPANY (salary);
```

Now, let's list down all the indices available in COMPANY table using **.indices** command as follows:

```
sqlite> .indices COMPANY
```

This will produce the following result, where *sqlite_autoindex_COMPANY_1* is an implicit index which got created when the table itself was created.

```
salary_index

sqlite_autoindex_COMPANY_1
```

You can list down all the indexes database wide as follows:

```
sqlite> SELECT * FROM sqlite_master WHERE type = 'index';
```

(   ) chinook.db, northwind.db, hr.db   .
primary key, foreign key   index   index
                         .   index   .

tutorialspoint
SIMPLY EASY LEARNING

The DROP INDEX Command

An index can be dropped using SQLite **DROP** command. Care should be taken when dropping an index because performance may be slowed or improved.

The basic syntax is as follows:

```
DROP INDEX index_name;
```

You can use the following statement to delete a previously created index.

```
sqlite> DROP INDEX salary_index;
```

When Should Indexes Be Avoided?

Although indexes are intended to enhance the performance of a database, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered.

Indexes should not be used in –

- Small tables.

- Tables that have frequent, large batch update or insert operations.

- Columns that contain a high number of NULL values.

- Columns that are frequently manipulated.

.

pragma   .index_list(   );

                                              .

pragma   .index_info(   );

(   ) index_list, index_info, table_info, ...   pragma   , pragma

                                              .

attach   .

The "INDEXED BY index-name" clause specifies that the named index must be used in order to look up values on the preceding table.

If index-name does not exist or cannot be used for the query, then the preparation of the SQLite statement fails.

The "NOT INDEXED" clause specifies that no index shall be used when accessing the preceding table, including implied indices created by UNIQUE and PRIMARY KEY constraints.

However, the INTEGER PRIMARY KEY can still be used to look up entries even when "NOT INDEXED" is specified.

Syntax

Following is the syntax for INDEXED BY clause and it can be used with DELETE, UPDATE or SELECT statement.

```
SELECT|DELETE|UPDATE column1, column2...

INDEXED BY (index_name)

table_name

WHERE (CONDITION);
```

Example

Consider table COMPANY. We will create an index and use it for performing INDEXED BY operation.

```
sqlite> CREATE INDEX salary_index ON COMPANY(salary);

sqlite>
```

Now selecting the data from table COMPANY you can use INDEXED BY clause as follows:

```
sqlite> SELECT * FROM COMPANY INDEXED BY salary_index WHERE salary > 5000;
```

This will produce the following result.

```
ID          NAME        AGE         ADDRESS     SALARY

----------  ----------  ----------  ----------  ----------

7           James       24          Houston     10000.0

2           Allen       25          Texas       15000.0

1           Paul        32          California  20000.0

3           Teddy       23          Norway      20000.0

6           Kim         22          South-Hall  45000.0
```

| 4 | Mark | 25 | Rich-Mond | 65000.0 |
| 5 | David | 27 | Texas | 85000.0 |

35. SQLite – ALTER TABLE Command

There is no other operation supported by ALTER TABLE command in SQLite except renaming a table and adding a column in an existing table.

Syntax

Following is the basic syntax of **ALTER TABLE** to RENAME an existing table.

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

Following is the basic syntax of **ALTER TABLE** to add a new column in an existing table.

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

Example

Consider the COMPANY table with the following records.

5	David	27	Texas	85000.0
6	Kim	22	South-Hall	45000.0
7	James	24	Houston	10000.0

Now, let's try to rename this table using ALTER TABLE statement as follows:

```
sqlite> ALTER TABLE COMPANY RENAME TO OLD_COMPANY;
```

The above SQLite statement will rename COMPANY table to OLD_COMPANY. Now, let's try to add a new column in OLD_COMPANY table as follows:

```
sqlite> ALTER TABLE OLD_COMPANY ADD COLUMN SEX char(1);
```

COMPANY table is now changed and following will be the output from SELECT statement.

```
ID          NAME        AGE         ADDRESS     SALARY      SEX

----------  ----------  ----------  ----------  ----------  ---

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
```

It should be noted that newly added column is filled with NULL values.

(   ) chinook.db   customers   email   unique
             . sqlite   alter table

                                        .

1. customers
              . (customers   create table
                  .)
2. insert into   select * from customers;   customers
                              .
3. customers   .
4.   customers   .

DBMS   truncate table   ;   SQLite
. truncate table   where   delete
delete   .

SQLite **DELETE** command to delete complete data from an existing table, though it is recommended to use DROP TABLE command to drop the complete table and re-create it once again.

Syntax

Following is the basic syntax of DELETE command.

```
sqlite> DELETE FROM table_name;
```
truncate table   ;

Following is the basic syntax of DROP TABLE.

```
sqlite> DROP TABLE table_name;
```

If you are using DELETE ~~TABLE~~ command to delete all the records, it is recommended to use **VACUUM** command to clear unused space.

Example

Consider COMPANY table with the following records.

```
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
```

Following is the example to truncate the above table:

```
SQLite> DELETE FROM COMPANY;
SQLite> VACUUM;
```

SQLite
.   DBMS   .
DBMS   .

(   ) testDB.db   backup   .

1. testDB.db   .
2. testDB.db   company   .
3. testDB.db   .
4. testDB.db   vacuum   .
5. testDB.db   .

(   ) vacuum   attach   . sqlite
DB   .
(   ) vacuum   .

select   . select
                                    .   , select
                         (virtual table)   .
    (view)   .

                         schema diagram
    .   .
                ,
                         .   /
      .

       DML   . SQLite
DML   .

• Summarize data from various tables, which can be used to generate reports.

<mark>SQLite views are read-only and thus you may not be able to execute a DELETE, INSERT, or UPDATE statement on a view.</mark> However, you can create a trigger on a view that fires on an attempt to DELETE, INSERT, or UPDATE a view and do what you need in the body of the trigger.

Creating Views

SQLite views are created using the **CREATE VIEW** statement. SQLIte views can be created from a single table, multiple tables, or another view.

Following is the basic CREATE VIEW syntax.

```
CREATE [TEMP | TEMPORARY] VIEW view_name AS

SELECT column1, column2.....

FROM table_name

WHERE [condition];
```

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query. <mark>If the optional TEMP or TEMPORARY keyword is present, the view will be created in the temp database.</mark>

Example

Consider COMPANY table with the following records.

```
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
```

Following is an example to create a view from COMPANY table. This view will be used to have only a few columns from COMPANY table.

```
sqlite> CREATE VIEW COMPANY_VIEW AS
SELECT ID, NAME, AGE
FROM  COMPANY;
```

You can now query COMPANY_VIEW in a similar way as you query an actual table. Following is an example:

```
sqlite> SELECT * FROM COMPANY_VIEW;
```

This will produce the following result.

```
ID          NAME        AGE
----------  ----------  ----------
1           Paul        32
2           Allen       25
3           Teddy       23
4           Mark        25
5           David       27
6           Kim         22
7           James       24
```

Dropping Views

To drop a view, simply use the DROP VIEW statement with the **view_name**. The basic DROP VIEW syntax is as follows:

```
sqlite> DROP VIEW view_name;
```

The following command will delete COMPANY_VIEW view, which we created in the last section.

```
sqlite> DROP VIEW COMPANY_VIEW;
```

(   )   ,   ,   ,   select
                select   .

(1) chinook.db
    .
(2) scott.db   view   .
(3) Northwind.db   10   view
    .

chinook.db   ?

1. 50   1, 2, 3   track   1.00   1   .

1) invoices   invoice   . (insert into   )
2) invoice_items   3   . (insert into   2)

   4   insert into   ?

2. 50   2, 3   .

1) invoice_items   2, 3   . (delete   )
2) invoices   Total   . (update   )

                                                        ?
SQL   (transaction)   .
       .   SQL   . (all- or-
nothing)

Transactions have the following four standard properties, usually referred to by the acronym ACID.

- **Atomicity**: Er   :
  successfully;   .   .
  previous operations are rolled back to their former state.

- **Consistency**:   :
  successfully c   .

- **Isolation**: Er   :   .
  each other.
                                                    (   )   .

- **Durability**: E   :
  case of a syste   .

Transaction Control TCL

Following are the commands used to control transactions:

- **BEGIN TRANSACTION**: To start a transaction.

- **COMMIT**: To save the changes, alternatively you can use END TRANSACTION command.

- **ROLLBACK**: To roll back the changes.

Transactional control commands are only used with DML commands INSERT, UPDATE, and DELETE. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

BEGIN TRANSACTION Command

Transactions can be started using BEGIN TRANSACTION or simply BEGIN command. Such transactions usually persist until the next COMMIT or ROLLBACK command is encountered. However, a transaction will also ROLLBACK if the database is closed or if an error occurs. Following is the simple syntax to start a transaction.

```
BEGIN;


or


BEGIN TRANSACTION;
```

COMMIT Command

COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

Following is the syntax for COMMIT command.

```
COMMIT;


or


END TRANSACTION;
```

ROLLBACK Command

ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.

ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

Following is the syntax for ROLLBACK command.

```
ROLLBACK;
```

Example

Consider COMPANY table with the following records.

```
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

Now, let's start a transaction and delete records from the table having age = 25. Then, use ROLLBACK command to undo all the changes.

```
sqlite> BEGIN;
sqlite> DELETE FROM COMPANY WHERE AGE = 25;
sqlite> ROLLBACK;
```

Now, if you check COMPANY table, it still has the following records:

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

Let's start another transaction and delete records from the table having age = 25 and finally we use COMMIT command to commit all the changes.

```
sqlite> BEGIN;
sqlite> DELETE FROM COMPANY WHERE AGE = 25;
sqlite> COMMIT;
```

If you now check COMPANY table still has the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	Paul	32	California	20000.0

( ) chinook.db .
?

1. 50 1, 2, 3, 4, 5 1.00 1 .
.

2. 50 . .

select, insert, update, delete   select   (,
subquery, inner query, nested query)   .   (
       , main query)   .   .
                       .

                                                                          .
= , < , > , > = , < = , in   .
                                                                                   .
                                                2
                                                                       2
            in, not in, exists   . exists
    1   True   False   .

- Subqueries must be enclosed within parentheses.

- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.

- An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a subquery.

- Subqueries that return more than one row can only be used with multiple value operators, such as the IN operator.

- BETWEEN operator cannot be used with a subquery; however, BETWEEN can be used within the subquery.

Subqueries with SELECT Statement

between (   ) and (   )
                  .

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows:

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
       (SELECT column_name [, column_name ]
       FROM table1 [, table2 ]
       [WHERE])
```

Example

Consider COMPANY table with the following records.

```
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
```

Now, let us check the following sub-query with SELECT statement.

```
sqlite> SELECT *

     FROM COMPANY

     WHERE ID IN (SELECT ID

             FROM COMPANY

             WHERE SALARY > 45000) ;
```

This will produce the following result.

```
(           )
1. scott.db      JONES                             ?
2. chinookdb      1                                         .
3. chinook.db      400                                      .
4. hr.db       Neena Kochhar                          .
```

Subqueries with INSERT Statement

Subqueries can also be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date, or number functions.

The basic syntax is as follows:

```
INSERT INTO table_name [ (column1 [, column2 ]) ]

        SELECT [ *|column1 [, column2 ]

        FROM table1 [, table2 ]

        [ WHERE VALUE OPERATOR ]
```

Example

Consider a table COMPANY_BKP with similar structure as COMPANY table and can be created using the same CREATE TABLE using COMPANY_BKP as the table name. To copy the complete COMPANY table into COMPANY_BKP, following is the syntax:

```
sqlite> INSERT INTO COMPANY_BKP
    SELECT * FROM COMPANY
    WHERE ID IN (SELECT ID
                FROM COMPANY) ;
```

Subqueries with UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows:

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
    (SELECT COLUMN_NAME
    FROM TABLE_NAME)
    [ WHERE) ]
```

Example

Assuming, we have COMPANY_BKP table available which is a backup of COMPANY table.

Following example updates SALARY by 0.50 times in COMPANY table for all the customers, whose AGE is greater than or equal to 27.

```
sqlite> UPDATE COMPANY
    SET SALARY = SALARY * 0.50
    WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
                WHERE AGE >= 27 );
```

27   50%   where
    .

where age > = 27

(   )
1. (scott.db) sales   10%   . (
                      rollback   .)

```
4          Mark          25          Rich-Mond    65000.0
```

5	David	27	Texas	42500.0
6	Kim	22	South-Hall	45000.0
7	James	24	Houston	10000.0

Subqueries with DELETE Statement

Subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows:

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
    (SELECT COLUMN_NAME
    FROM TABLE_NAME)
    [ WHERE) ]
```

Example

Assuming, we have COMPANY_BKP table available which is a backup of COMPANY table.

Following example deletes records from COMPANY table for a 27
is greater than or equal to 27.

```
sqlite> DELETE FROM COMPANY
      WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
                WHERE AGE > 27 );
```

This will impact two rows and finally COMPANY table will have the following records.

40. SQLite – AUTOINCREMENT

SQLite **AUTOINCREMENT** is a keyword used for auto incrementing a value of a field in the table. We can auto increment a field value by using **AUTOINCREMENT** keyword when creating a table with specific column name to auto increment it.

The keyword **AUTOINCREMENT** can be used with INTEGER field only.

Syntax

The basic usage of **AUTOINCREMENT** keyword is as follows:

```
CREATE TABLE table_name(
    column1 INTEGER AUTOINCREMENT,
    column2 datatype,
    column3 datatype,
    .....
    columnN datatype,
);
```

Example

Consider COMPANY table to be created as follows:

```
sqlite> CREATE TABLE COMPANY(
    ID INTEGER PRIMARY KEY    AUTOINCREMENT,
    NAME          TEXT      NOT NULL,
    AGE           INT       NOT NULL,
    ADDRESS       CHAR(50),
    SALARY        REAL
);
```

Now, insert the following records into table COMPANY:

```
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'Paul', 32, 'California', 20000.00 );


INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ('Allen', 25, 'Texas', 15000.00 );


INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
```

```
VALUES ('Teddy', 23, 'Norway', 20000.00 );


INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)

VALUES ( 'Mark', 25, 'Rich-Mond ', 65000.00 );


INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)

VALUES ( 'David', 27, 'Texas', 85000.00 );



INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)

VALUES ( 'Kim', 22, 'South-Hall', 45000.00 );


INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)

VALUES ( 'James', 24, 'Houston', 10000.00 );
```

This will insert 7 tuples into the table COMPANY and COMPANY will have the following records:

```
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
```

41. SQLite – Injection

If you take user input through a webpage and insert it into a SQLite database there's a chance that you have left yourself wide open for a security issue known as SQL Injection. In this chapter, you will learn how to help prevent this from happening and help you secure your scripts and SQLite statements.

Injection usually occurs when you ask a user for input, like their name, and instead of a name they give you a SQLite statement that you will unknowingly run on your database.

Never trust user provided data, process this data only after validation; as a rule, this is done by pattern matching. In the following example, the username is restricted to alphanumerical chars plus underscore and to a length between 8 and 20 chars - modify these rules as needed.

```
if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches)){

    $db = new SQLiteDatabase('filename');

    $result = @$db->query("SELECT * FROM users WHERE username=$matches[0]");

}else{

    echo "username not accepted";

}
```

To demonstrate the problem, consider this excerpt:

```
$name = "Qadir'; DELETE FROM users;";
@$db->query("SELECT * FROM users WHERE username='{$name}'");
```

The function call is supposed to retrieve a record from the users table where the name column matches the name specified by the user. Under normal circumstances, **$name** would only contain alphanumeric characters and perhaps spaces, such as the string ilia. However in this case, by appending an entirely new query to $name, the call to the database turns into a disaster: the injected DELETE query removes all records from users.

There are databases interfaces which do not permit query stacking or executing multiple queries in a single function call. If you try to stack queries, the call fails but SQLite and PostgreSQL, happily perform stacked queries, executing all of the queries provided in one string and creating a serious security problem.

Preventing SQL Injection

You can handle all escape characters smartly in scripting languages like PERL and PHP. Programming language PHP provides the function **string sqlite_escape_string()** to escape input characters that are special to SQLite.

```
if (get_magic_quotes_gpc())
{
  $name = sqlite_escape_string($name);
}
$result = @$db->query("SELECT * FROM users WHERE username='{$name}'");
```

Although the encoding makes it safe to insert the data, it will render simple text comparisons and **LIKE** clauses in your queries unusable for the columns that contain the binary data.

Note: **addslashes()** should NOT be used to quote your strings for SQLite queries; it will lead to strange results when retrieving your data.

SQLite statement can be preceded by the keyword "EXPLAIN" or by the phrase "EXPLAIN QUERY PLAN" used for describing the details of a table.

Either modification causes the SQLite statement to behave as a query and to return information about how the SQLite statement would have operated if the EXPLAIN keyword or phrase had been omitted.

- The output from EXPLAIN and EXPLAIN QUERY PLAN is intended for interactive analysis and troubleshooting only.

- The details of the output format are subject to change from one release of SQLite to the next.

- Applications should not use EXPLAIN or EXPLAIN QUERY PLAN since their exact behavior is variable and only partially documented.

Syntax

Syntax for **EXPLAIN** is as follows:

```
EXPLAIN [SQLite Query]
```

Syntax for **EXPLAIN QUERY PLAN** is as follows:

```
EXPLAIN  QUERY PLAN [SQLite Query]
```

Example

Consider COMPANY table with the following records:

```
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
```

Now, let us check the following sub-query with SELECT statement:

```
sqlite> EXPLAIN SELECT *  FROM COMPANY  WHERE Salary >= 20000;
```

This will produce the following result:

addr	opcode	p1	p2	p3
0	Goto	0	19	
1	Integer	0	0	
2	OpenRead	0	8	
3	SetNumColu	0	5	
4	Rewind	0	17	
5	Column	0	4	
6	RealAffini	0	0	
7	Integer	20000	0	
8	Lt	357	16	collseq(BI
9	Rowid	0	0	
10	Column	0	1	
11	Column	0	2	
12	Column	0	3	
13	Column	0	4	
14	RealAffini	0	0	
15	Callback	5	0	
16	Next	0	5	
17	Close	0	0	
18	Halt	0	0	
19	Transactio	0	0	
20	VerifyCook	0	38	
21	Goto	0	1	
22	Noop	0	0	

Now, let us check the following **Explain Query Plan** with SELECT statement:

```
SQLite> EXPLAIN QUERY PLAN SELECT * FROM COMPANY WHERE Salary >= 20000;
order       from        detail
----------  ----------  -------------
0           0           TABLE COMPANY
```

VACUUM command cleans the main database by copying its contents to a temporary database file and reloading the original database file from the copy. This eliminates free pages, aligns table data to be contiguous, and otherwise cleans up the database file structure.

VACUUM command may change the ROWID of entries in tables that do not have an explicit INTEGER PRIMARY KEY. The VACUUM command only works on the main database. It is not possible to VACUUM an attached database file.

VACUUM command will fail if there is an active transaction. VACUUM command is a no-op for in-memory databases. As the VACUUM command rebuilds the database file from scratch, VACUUM can also be used to modify many database-specific configuration parameters.

Manual VACUUM

Following is a simple syntax to issue a VACUUM command for the whole database from command prompt:

```
$sqlite3 database_name "VACUUM;"
```

You can run VACUUM from SQLite prompt as well as follows:

```
sqlite> VACUUM;
```

You can also run VACUUM on a particular table as follows:

```
sqlite> VACUUM table_name;
```

Auto-VACCUM

SQLite Auto-VACUUM does not do the same as VACUUM rather it only moves free pages to the end of the database thereby reducing the database size. By doing so it can significantly fragment the database while VACUUM ensures defragmentation. Hence, Auto-VACUUM just keeps the database small.

You can enable/disable SQLite auto-vacuuming by the following pragmas running at SQLite prompt:

```
sqlite> PRAGMA auto_vacuum = NONE;   -- 0 means disable auto vacuum

sqlite> PRAGMA auto_vacuum = FULL;   -- 1 means enable full auto vacuum

sqlite> PRAGMA auto_vacuum = INCREMENTAL;   -- 2 means enable incremental vacuum
```

You can run the following command from the command prompt to check the auto-vacuum setting:

```
$sqlite3 database_name "PRAGMA auto_vacuum;"
```

,   ISO   ,
,   ,   ,
.

Sr. No.	Function	Example
1	date(timestring, modifiers...)	This returns the date in this format: YYYY-MM-DD
2	time(timestring, modifiers...)	This returns the time as HH:MM:SS
3	datetime(timestring, modifiers...)	This returns YYYY-MM-DD HH:MM:SS
4	julianday(timestring, modifiers...)	This returns the number of days since noon in Greenwich on November 24, 4714 B.C.
	strftime(formatstring, timestring)	This returns the date formatted according to the format string specified as the first argument formatted as per formatters explained below.

All the above five date and time functions take a time string as an argument. The time string is followed by zero or more modifiers. The strftime() function also takes a format string as its first argument. Following section will give you detail on different types of time strings and modifiers.

Time Strings

A time string can be in any of the following formats:

Sr. No.	Time String	Example
1	YYYY-MM-DD	2010-12-30
2	YYYY-MM-DD HH:MM	2010-12-30 12:10
3	YYYY-MM-DD HH:MM:SS.SSS	2010-12-30 12:10:04.100
4	MM-DD-YYYY HH:MM	30-12-2010 12:10
5	HH:MM	12:10
6	YYYY-MM-DD**T**HH:MM	2010-12-30 12:10

7	HH:MM:SS	12:10:01
8	YYYYMMDD HHMMSS	20101230 121001
9	now	2013-05-07

You can use the "T" as a literal character separating the date and the time.

Modifiers

The time string can be followed by zero or more modifiers that will alter date and/or time returned by any of the above five functions. Modifiers are applied from the left to right.

Following modifiers are available in SQLite:

- NNN days
- NNN hours
- NNN minutes
- NNN.NNNN seconds
- NNN months
- NNN years
- start of month
- start of year
- start of day
- weekday N
- unixepoch
- localtime
- utc

Formatters

SQLite provides a very handy function **strftime()** to format any date and time. You can use the following substitutions to format your date and time.

Substitution	Description
%d	Day of month, 01-31
%f	Fractional seconds, SS.SSS
%H	Hour, 00-23

%j	Day of year, 001-366
%J	Julian day number, DDDD.DDDD
%m	Month, 00-12
%M	Minute, 00-59
%s	Seconds since 1970-01-01 ← unixepoch, unixtimestamp
%S	Seconds, 00-59
%w	Day of week, 0-6 (0 is Sunday)
%W	Week of year, 01-53
%Y	Year, YYYY
%%	% symbol

Examples

Let's try various examples now using SQLite prompt. Following command computes the current date.

```
sqlite> SELECT date('now');
2013-05-07
```

time('now'), date('now'), datetime('now')
current_time, current_date, current_timestamp
. UTC
(local time) .

Following command computes th

```
sqlite> SELECT date('now','start of month','+1 month','-1 day');
2013-05-31
```

Following command computes the date and time for a given UNIX timestamp 1092941466.

```
sqlite> SELECT datetime(1092941466, 'unixepoch');
2004-08-19 18:51:06
```

Following command computes the date and time for a given UNIX timestamp 1092941466 and compensates for your local timezone.

```
sqlite> SELECT datetime(1092941466, 'unixepoch', 'localtime');
2004-08-19 13:51:06
```

Following command computes the current UNIX timestamp.

```
sqlite> SELECT strftime('%s','now');
1393348134
```

( ) select , select where , insert, update
. scott.db .

1. .
2. .

(   ) chinook.db

1.   .
2. 2013   11   12   2   .
3. 2013   11   12   2   .

of Independence.

```
sqlite> SELECT julianday('now') - julianday('1776-07-04');

86798.7094695023
```

Following command computes the number of seconds since a particular moment in 2004.

```
sqlite> SELECT strftime('%s','now') - strftime('%s','2004-01-01 02:34:56');
295001572
```

julianday(), strftime('%s', timestring)   .
julianday()   .

```
2013-10-01
```

,   strftime('%m', timestring), strftime('%d',
timestring)   .   cast(as decimal)
              .

        3   .

```
select strftime('%Y', invoicedate) as year, sum(total) from invoice
where cast(strftime('%m', invoicedate) as integer) = 3
group by year;
```

cf. integer, decimal, float, real, text etc.

```
05:00:00
```

```
sqlite>  SELECT time('12:00', 'utc');
19:00:00
```

select datetime('now'), datetime('now', 'localtime'), datetime('now', 'utc');

'now'   utc   . 'localtime' modifier   utc
                              , 'utc' modifier
      utc   .   'utc' modifier
            .

SQLite has many built-in functions to perform processing on string or numeric data. Following is the list of few useful SQLite built-in functions and all are case in-sensitive which means you can use these functions either in lower-case form or in upper-case or in mixed form. For more details, you can check official documentation for SQLite.

Sr. No.	Function & Description
1	**SQLite COUNT Function** The SQLite COUNT aggregate function is used to count the number of rows in a database table.
2	**SQLite MAX Function** The SQLite MAX aggregate function allows us to select the highest (maximum) value for a certain column.
3	**SQLite MIN Function** The SQLite MIN aggregate function allows us to select the lowest (minimum) value for a certain column.
4	**SQLite AVG Function** The SQLite AVG aggregate function selects the average value for certain table column.
5	**SQLite SUM Function** The SQLite SUM aggregate function allows selecting the total for a numeric column.
6	**SQLite RANDOM Function** The SQLite RANDOM function returns a pseudo-random integer between -9223372036854775808 and +9223372036854775807.

7	**SQLite ABS Function** The SQLite ABS function returns the absolute value of the numeric argument.
8	**SQLite UPPER Function** The SQLite UPPER function converts a string into upper-case letters.
9	**SQLite LOWER Function** The SQLite LOWER function converts a string into lower-case letters.
10	**SQLite LENGTH Function** The SQLite LENGTH function returns the length of a string.
11	**SQLite sqlite_version Function** The SQLite sqlite_version function returns the version of the SQLite library.

Before we start giving examples on the above-mentioned functions, consider COMPANY table with the following records.

```
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
```

SQLite COUNT Function

SQLite COUNT aggregate function is used to count the number of rows in a database table. Following is an example:

```
sqlite> SELECT count(*) FROM COMPANY;
```

The above SQLite SQL statement will produce the following result:

```
count(*)
----------
7
```

SQLite MAX Function

SQLite MAX aggregate function allows us to select the highest (maximum) value for a certain column. Following is an example:

```
sqlite> SELECT max(salary) FROM COMPANY;
```

The above SQLite SQL statement will produce the following result.

```
max(salary)
-----------
85000.0
```

SQLite MIN Function

SQLite MIN aggregate function allows us to select the lowest (minimum) value for a certain column. Following is an example:

```
sqlite> SELECT min(salary) FROM COMPANY;
```

The above SQLite SQL statement will produce the following result.

```
min(salary)
-----------
10000.0
```

SQLite AVG Function

SQLite AVG aggregate function selects the average value for a certain table column. Following is an example:

```
sqlite> SELECT avg(salary) FROM COMPANY;
```

The above SQLite SQL statement will produce the following result.

```
avg(salary)
----------------
37142.8571428572
```

SQLite SUM Function

SQLite SUM aggregate function allows selecting the total for a numeric column. Following is an example:

```
sqlite> SELECT sum(salary) FROM COMPANY;
```

The above SQLite SQL statement will produce the following result.

```
sum(salary)
-----------
260000.0
```

SQLite RANDOM Function

SQLite RANDOM function returns a pseudo-random integer between -9223372036854775808 and +9223372036854775807. Following is an example:

```
sqlite> SELECT random() AS Random;
```

The above SQLite SQL statement will produce the following result.

```
Random
------------------
5876796417670984050
```

SQLite ABS Function

SQLite ABS function returns the absolute value of the numeric argument. Following is an example:

```
sqlite> SELECT abs(5), abs(-15), abs(NULL), abs(0), abs("ABC");
```

The above SQLite SQL statement will produce the following result.

```
abs(5)      abs(-15)    abs(NULL)   abs(0)      abs("ABC")
----------  ----------  ----------  ----------  ----------
5           15                      0           0.0
```

SQLite UPPER Function

SQLite UPPER function converts a string into upper-case letters. Following is an example:

```
sqlite> SELECT upper(name) FROM COMPANY;
```

The above SQLite SQL statement will produce the following result.

```
upper(name)
-----------
PAUL
ALLEN
TEDDY
MARK
DAVID
KIM
JAMES
```

SQLite LOWER Function

SQLite LOWER function converts a string into lower-case letters. Following is an example:

```
sqlite> SELECT lower(name) FROM COMPANY;
```

The above SQLite SQL statement will produce the following result.

```
lower(name)
-----------
paul
allen
teddy
mark
david
kim
james
```

SQLite LENGTH Function

SQLite LENGTH function returns the length of a string. Following is an example:

```
sqlite> SELECT name, length(name) FROM COMPANY;
```

The above SQLite SQL statement will produce the following result.

```
NAME        length(name)
----------  ------------
Paul        4
Allen       5
```

```
Teddy        5
Mark         4
David        5
Kim          3
James        5
```

SQLite sqlite_version Function

SQLite sqlite_version function returns the version of the SQLite library. Following is an example:

```
sqlite> SELECT sqlite_version() AS 'SQLite Version';
```

The above SQLite SQL statement will produce the following result.

```
SQLite Version
--------------
3.6.20
```

SQLite Interfaces

In this chapter, you will learn how to use SQLite in C/C++ programs.

Installation

Before you start using SQLite in our C/C++ programs, you need to make sure that you have SQLite library set up on the machine. You can check SQLite Installation chapter to understand the installation process.

C/C++ Interface APIs

Following are important C/C++ SQLite interface routines, which can suffice your requirement to work with SQLite database from your C/C++ program. If you are looking for a more sophisticated application, then you can look into SQLite official documentation.

Sr. No.	API & Description
1	**sqlite3_open(const char *filename, sqlite3 **ppDb)** This routine opens a connection to an SQLite database file and returns a database connection object to be used by other SQLite routines. If the *filename* argument is NULL or ':memory:', sqlite3_open() will create an in-memory database in RAM that lasts only for the duration of the session. If the filename is not NULL, sqlite3_open() attempts to open the database file by using its value. If no file by that name exists, sqlite3_open() will open a new database file by that name.
2	**sqlite3_exec(sqlite3*, const char *sql, sqlite_callback, void *data, char **errmsg)** This routine provides a quick, easy way to execute SQL commands provided by sql argument, which can consist of more than one SQL command. Here, the first argument *sqlite3* is an open database object, *sqlite_callback* is a callback for which *data* is the 1st argument and errmsg will be returned to capture any error raised by the routine. The sqlite3_exec() routine parses and executes every command given in the **sql** argument until it reaches the end of the string or encounters an error.

3	**sqlite3_close(sqlite3*)** This routine closes a database connection previously opened by a call to sqlite3_open(). All prepared statements associated with the connection should be finalized prior to closing the connection. If any queries remain that have not been finalized, sqlite3_close() will return SQLITE_BUSY with the error message Unable to close due to unfinalized statements.

Connect to Database

Following C code segment shows how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned.

```c
#include <stdio.h>
#include <sqlite3.h>


int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;


   rc = sqlite3_open("test.db", &db);


   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      return(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }
   sqlite3_close(db);
}
```

Now, let's compile and run the above program to create our database **test.db** in the current directory. You can change your path as per your requirement.

```
$gcc test.c -l sqlite3
$./a.out
Opened database successfully
```

If you are going to use C++ source code, then you can compile your code as follows:

```
$g++ test.c -l sqlite3
```

Here, we are linking our program with sqlite3 library to provide required functions to C program. This will create a database file test.db in your directory and you will have the following result.

```
-rwxr-xr-x. 1 root root 7383 May  8 02:06 a.out
-rw-r--r--. 1 root root  323 May  8 02:05 test.c
-rw-r--r--. 1 root root    0 May  8 02:06 test.db
```

Create a Table

Following C code segment will be used to create a table in the previously created database.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *NotUsed, int argc, char **argv, char **azColName){
   int i;
   for(i=0; i<argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int  rc;
   char *sql;

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
```

```
      return(0);
   }else{
      fprintf(stdout, "Opened database successfully\n");
   }


   /* Create SQL statement */
   sql = "CREATE TABLE COMPANY("  \
         "ID INT PRIMARY KEY     NOT NULL," \
         "NAME           TEXT    NOT NULL," \
         "AGE            INT     NOT NULL," \
         "ADDRESS        CHAR(50)," \
         "SALARY         REAL );";


   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
   if( rc != SQLITE_OK ){
   fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Table created successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

When the above program is compiled and executed, it will create COMPANY table in your test.db and the final listing of the file will be as follows:

```
-rwxr-xr-x. 1 root root 9567 May  8 02:31 a.out
-rw-r--r--. 1 root root 1207 May  8 02:31 test.c
-rw-r--r--. 1 root root 3072 May  8 02:31 test.db
```

INSERT Operation

Following C code segment shows how you can create records in COMPANY table created in the above example:

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>
```

143

```
static int callback(void *NotUsed, int argc, char **argv, char **azColName){
   int i;
   for(i=0; i<argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      return(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create SQL statement */
   sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "  \
         "VALUES (1, 'Paul', 32, 'California', 20000.00 ); " \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "  \
         "VALUES (2, 'Allen', 25, 'Texas', 15000.00 ); "      \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)" \
         "VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );" \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)" \
         "VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
   if( rc != SQLITE_OK ){
```

```
        fprintf(stderr, "SQL error: %s\n", zErrMsg);

        sqlite3_free(zErrMsg);

    }else{

        fprintf(stdout, "Records created successfully\n");

    }

    sqlite3_close(db);

    return 0;

}
```

When the above program is compiled and executed, it will create the given records in COMPANY table and will display the following two lines:

```
Opened database successfully

Records created successfully
```

SELECT Operation

Before proceeding with actual example to fetch records, let us look at some detail about the callback function, which we are using in our examples. This callback provides a way to obtain results from SELECT statements. It has the following declaration:

```
typedef int (*sqlite3_callback)(

void*,     /* Data provided in the 4th argument of sqlite3_exec() */

int,       /* The number of columns in row */

char**,    /* An array of strings representing fields in the row */

char**     /* An array of strings representing column names */

);
```

If the above callback is provided in sqlite_exec() routine as the third argument, SQLite will call this callback function for each record processed in each SELECT statement executed within the SQL argument.

Following C code segment shows how you can fetch and display records from the COMPANY table created in the above example.

```
#include <stdio.h>

#include <stdlib.h>

#include <sqlite3.h>


static int callback(void *data, int argc, char **argv, char **azColName){

    int i;

    fprintf(stderr, "%s: ", (const char*)data);

    for(i=0; i<argc; i++){

        printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
```

145

```
   }
   printf("\n");
   return 0;
}


int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";


   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      return(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }
   /* Create SQL statement */
   sql = "SELECT * from COMPANY";


   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

When the above program is compiled and executed, it will produce the following result.

```
Opened database successfully
Callback function called: ID = 1
```

```
NAME = Paul

AGE = 32

ADDRESS = California

SALARY = 20000.0


Callback function called: ID = 2

NAME = Allen

AGE = 25

ADDRESS = Texas

SALARY = 15000.0


Callback function called: ID = 3

NAME = Teddy

AGE = 23

ADDRESS = Norway

SALARY = 20000.0


Callback function called: ID = 4

NAME = Mark

AGE = 25

ADDRESS = Rich-Mond

SALARY = 65000.0


Operation done successfully
```

UPDATE Operation

Following C code segment shows how we can use UPDATE statement to update any record and then fetch and display updated records from the COMPANY table.

```c
#include <stdio.h>

#include <stdlib.h>

#include <sqlite3.h>


static int callback(void *data, int argc, char **argv, char **azColName){
    int i;
    fprintf(stderr, "%s: ", (const char*)data);
    for(i=0; i<argc; i++){
        printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
```

```
   }
   printf("\n");
   return 0;
}


int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";


   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      return(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }


   /* Create merged SQL statement */
   sql = "UPDATE COMPANY set SALARY = 25000.00 where ID=1; " \
         "SELECT * from COMPANY";


   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

When the above program is compiled and executed, it will produce the following result.

```
Opened database successfully

Callback function called: ID = 1

NAME = Paul

AGE = 32

ADDRESS = California

SALARY = 25000.0


Callback function called: ID = 2

NAME = Allen

AGE = 25

ADDRESS = Texas

SALARY = 15000.0


Callback function called: ID = 3

NAME = Teddy

AGE = 23

ADDRESS = Norway

SALARY = 20000.0


Callback function called: ID = 4

NAME = Mark

AGE = 25

ADDRESS = Rich-Mond

SALARY = 65000.0


Operation done successfully
```

DELETE Operation

Following C code segment shows how you can use DELETE statement to delete any record and then fetch and display the remaining records from the COMPANY table.

```
#include <stdio.h>

#include <stdlib.h>

#include <sqlite3.h>
```

```
static int callback(void *data, int argc, char **argv, char **azColName){
   int i;
   fprintf(stderr, "%s: ", (const char*)data);
   for(i=0; i<argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
   }
   printf("\n");
   return 0;
}


int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      return(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create merged SQL statement */
   sql = "DELETE from COMPANY where ID=2; " \
         "SELECT * from COMPANY";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
```

```
    }else{
        fprintf(stdout, "Operation done successfully\n");
    }
    sqlite3_close(db);
    return 0;
}
```

When the above program is compiled and executed, it will produce the following result.

```
Opened database successfully
Callback function called: ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0

Callback function called: ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

Callback function called: ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

In this chapter, you will learn how to use SQLite in Java programs.

Installation

Before you start using SQLite in your Java programs, you need to make sure that you have SQLite JDBC Driver and Java set up on the machine. You can check Java tutorial for Java installation on your machine. Now, let us check how to set up SQLite JDBC driver.

Step 1: Download latest version of sqlite-jdbc-(VERSION).jar from sqlite-jdbc repository.

Step 2: Add downloaded jar file sqlite-jdbc-(VERSION).jar in your class path, or you can use it along with -classpath option as explained in the following examples.

Following section assumes you have little knowledge about Java JDBC concepts. If you don't, then it is suggested to spent half an hour with JDBC Tutorial to become comfortable with the concepts explained below.

Connect to Database

Following Java programs show how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned.

```java
import java.sql.*;
public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMessage() );
      System.exit(0);
    }
    System.out.println("Opened database successfully");
  }
}
```

Now, let's compile and run the above program to create our database **test.db** in the current directory. You can change your path as per your requirement. We are assuming the current version of JDBC driver *sqlite-jdbc-3.7.2.jar* is available in the current path.

```
$javac SQLiteJDBC.java

$java -classpath ".:sqlite-jdbc-3.7.2.jar" SQLiteJDBC

Open database successfully
```

If you are going to use Windows machine, then you can compile and run your code as follows:

```
$javac SQLiteJDBC.java

$java -classpath ".;sqlite-jdbc-3.7.2.jar" SQLiteJDBC

Opened database successfully
```

Create a Table

Following Java program will be used to create a table in the previously created database.

```java
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      System.out.println("Opened database successfully");


      stmt = c.createStatement();
      String sql = "CREATE TABLE COMPANY " +
                   "(ID INT PRIMARY KEY     NOT NULL," +
                   " NAME           TEXT    NOT NULL, " +
                   " AGE            INT     NOT NULL, " +
                   " ADDRESS        CHAR(50), " +
                   " SALARY         REAL)";
      stmt.executeUpdate(sql);
      stmt.close();
```

153

```
      c.close();
   } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMessage() );
      System.exit(0);
   }
   System.out.println("Table created successfully");
  }
}
```

When the above program is compiled and executed, it will create COMPANY table in your **test.db** and final listing of the file will be as follows:

```
-rw-r--r--. 1 root root 3201128 Jan 22 19:04 sqlite-jdbc-3.7.2.jar
-rw-r--r--. 1 root root    1506 May  8 05:43 SQLiteJDBC.class
-rw-r--r--. 1 root root     832 May  8 05:42 SQLiteJDBC.java
-rw-r--r--. 1 root root    3072 May  8 05:43 test.db
```

INSERT Operation

Following Java program shows how to create records in the COMPANY table created in above example.

```java
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");


      stmt = c.createStatement();
      String sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
                   "VALUES (1, 'Paul', 32, 'California', 20000.00 );";
      stmt.executeUpdate(sql);
```

```
        sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
              "VALUES (2, 'Allen', 25, 'Texas', 15000.00 );";
        stmt.executeUpdate(sql);


        sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
              "VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );";
        stmt.executeUpdate(sql);


        sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
              "VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );";
        stmt.executeUpdate(sql);


        stmt.close();
        c.commit();
        c.close();
      } catch ( Exception e ) {
        System.err.println( e.getClass().getName() + ": " + e.getMessage() );
        System.exit(0);
      }
      System.out.println("Records created successfully");
    }
}
```

When the above program is compiled and executed, it will create the given records in the COMPANY table and will display the following two lines.

```
Opened database successfully

Records created successfully
```

SELECT Operation

Following Java program shows how to fetch and display records from the COMPANY table created in the above example.

```
import java.sql.*;


public class SQLiteJDBC
{
  public static void main( String args[] )
```

```
{
   Connection c = null;
   Statement stmt = null;
   try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
      while ( rs.next() ) {
         int id = rs.getInt("id");
         String  name = rs.getString("name");
         int age  = rs.getInt("age");
         String  address = rs.getString("address");
         float salary = rs.getFloat("salary");
         System.out.println( "ID = " + id );
         System.out.println( "NAME = " + name );
         System.out.println( "AGE = " + age );
         System.out.println( "ADDRESS = " + address );
         System.out.println( "SALARY = " + salary );
         System.out.println();
      }
      rs.close();
      stmt.close();
      c.close();
   } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMessage() );
      System.exit(0);
   }
   System.out.println("Operation done successfully");
   }
}
```

When the above program is compiled and executed, it will produce the following result.

```
Opened database successfully
```

tutorialspoint
SIMPLY EASY LEARNING

```
ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0


ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0


ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0


ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0


Operation done successfully
```

UPDATE Operation

Following Java code shows how to use UPDATE statement to update any record and then fetch and display the updated records from the COMPANY table.

```java
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
```

```
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");


      stmt = c.createStatement();
      String sql = "UPDATE COMPANY set SALARY = 25000.00 where ID=1;";
      stmt.executeUpdate(sql);
      c.commit();


      ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
      while ( rs.next() ) {
         int id = rs.getInt("id");
         String  name = rs.getString("name");
         int age  = rs.getInt("age");
         String  address = rs.getString("address");
         float salary = rs.getFloat("salary");
         System.out.println( "ID = " + id );
         System.out.println( "NAME = " + name );
         System.out.println( "AGE = " + age );
         System.out.println( "ADDRESS = " + address );
         System.out.println( "SALARY = " + salary );
         System.out.println();
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMessage() );
      System.exit(0);
    }
    System.out.println("Operation done successfully");
  }
}
```

When the above program is compiled and executed, it will produce the following result.

```
Opened database successfully
ID = 1
```

```
NAME = Paul

AGE = 32

ADDRESS = California

SALARY = 25000.0


ID = 2

NAME = Allen

AGE = 25

ADDRESS = Texas

SALARY = 15000.0


ID = 3

NAME = Teddy

AGE = 23

ADDRESS = Norway

SALARY = 20000.0


ID = 4

NAME = Mark

AGE = 25

ADDRESS = Rich-Mond

SALARY = 65000.0


Operation done successfully
```

DELETE Operation

Following Java code shows how to use DELETE statement to delete any record and then fetch and display the remaining records from the COMPANY table.

```
import java.sql.*;


public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
```

```
        Class.forName("org.sqlite.JDBC");
        c = DriverManager.getConnection("jdbc:sqlite:test.db");
        c.setAutoCommit(false);
        System.out.println("Opened database successfully");


        stmt = c.createStatement();
        String sql = "DELETE from COMPANY where ID=2;";
        stmt.executeUpdate(sql);
        c.commit();


        ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
        while ( rs.next() ) {
            int id = rs.getInt("id");
            String  name = rs.getString("name");
            int age  = rs.getInt("age");
            String  address = rs.getString("address");
            float salary = rs.getFloat("salary");
            System.out.println( "ID = " + id );
            System.out.println( "NAME = " + name );
            System.out.println( "AGE = " + age );
            System.out.println( "ADDRESS = " + address );
            System.out.println( "SALARY = " + salary );
            System.out.println();
        }
        rs.close();
        stmt.close();
        c.close();
    } catch ( Exception e ) {
        System.err.println( e.getClass().getName() + ": " + e.getMessage() );
        System.exit(0);
    }
    System.out.println("Operation done successfully");
  }
}
```

When the above program is compiled and executed, it will produce the following result.

```
Opened database successfully
ID = 1
NAME = Paul
```

```
AGE = 32
ADDRESS = California
SALARY = 25000.0


ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0


ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0


Operation done successfully
```

48. SQLite – PHP

In this chapter, you will learn how to use SQLite in PHP programs.

Installation

SQLite3 extension is enabled by default as of PHP 5.3.0. It's possible to disable it by using **--without-sqlite3** at compile time.

Windows users must enable **php_sqlite3.dll** in order to use this extension. This DLL is included with Windows distributions of PHP as of PHP 5.3.0.

For detailed installation instructions, kindly check our PHP tutorial and its official website.

PHP Interface APIs

Following are important PHP routines which can suffice your requirement to work with SQLite database from your PHP program. If you are looking for a more sophisticated application, then you can look into PHP official documentation.

Sr. No.	API & Description
1	**public void SQLite3::open (filename, flags, encryption_key)** Opens SQLite 3 Database. If the build includes encryption, then it will attempt to use the key. If the *filename* is given as **':memory:'**, SQLite3::open() will create an in-memory database in RAM that lasts only for the duration of the session. If the filename is actual device file name, SQLite3::open() attempts to open the database file by using its value. If no file by that name exists, then a new database file by that name gets created. Optional flags used to determine how to open the SQLite database. By default, open uses SQLITE3_OPEN_READWRITE | SQLITE3_OPEN_CREATE.
2	**public bool SQLite3::exec (string $query)** This routine provides a quick, easy way to execute SQL commands provided by sql argument, which can consist of more than one SQL command. This routine is used to execute a result-less query against a given database.
3	**public SQLite3Result SQLite3::query (string $query)**

	This routine executes an SQL query, returning an **SQLite3Result** object if the query returns results.
4	**public int SQLite3::lastErrorCode (void)** This routine returns the numeric result code of the most recent failed SQLite request.
5	**public string SQLite3::lastErrorMsg (void)** This routine returns English text describing the most recent failed SQLite request.
6	**public int SQLite3::changes (void)** This routine returns the number of database rows that were updated, inserted, or deleted by the most recent SQL statement.
7	**public bool SQLite3::close (void)** This routine closes a database connection previously opened by a call to SQLite3::open().
8	**public string SQLite3::escapeString (string $value)** This routine returns a string that has been properly escaped for safe inclusion in an SQL statement.

Connect to Database

Following PHP code shows how to connect to an existing database. If database does not exist, then it will be created and finally a database object will be returned.

```php
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
```

```
    $db = new MyDB();

    if(!$db){

        echo $db->lastErrorMsg();

    } else {

        echo "Opened database successfully\n";

    }

?>
```

Now, let's run the above program to create our database **test.db** in the current directory. You can change your path as per your requirement. If the database is successfully created, then it will display the following message:

```
Open database successfully
```

Create a Table

Following PHP program will be used to create a table in the previously created database.

```php
<?php
    class MyDB extends SQLite3
    {
        function __construct()
        {
            $this->open('test.db');
        }
    }
    $db = new MyDB();
    if(!$db){
        echo $db->lastErrorMsg();
    } else {
        echo "Opened database successfully\n";
    }

    $sql =<<<EOF
        CREATE TABLE COMPANY
        (ID INT PRIMARY KEY     NOT NULL,
        NAME            TEXT    NOT NULL,
        AGE             INT     NOT NULL,
        ADDRESS         CHAR(50),
        SALARY          REAL);
EOF;
```

164

```
    $ret = $db->exec($sql);
    if(!$ret){
        echo $db->lastErrorMsg();
    } else {
        echo "Table created successfully\n";
    }
    $db->close();
?>
```

When the above program is executed, it will create the COMPANY table in your **test.db** and it will display the following messages:

```
Opened database successfully

Table created successfully
```

INSERT Operation

Following PHP program shows how to create records in the COMPANY table created in the above example.

```
<?php
    class MyDB extends SQLite3
    {
        function __construct()
        {
            $this->open('test.db');
        }
    }
    $db = new MyDB();
    if(!$db){
        echo $db->lastErrorMsg();
    } else {
        echo "Opened database successfully\n";
    }

    $sql =<<<EOF
        INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
        VALUES (1, 'Paul', 32, 'California', 20000.00 );

        INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
```

```
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 );


      INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)

      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );


      INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)

      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );
EOF;


   $ret = $db->exec($sql);

   if(!$ret){

      echo $db->lastErrorMsg();

   } else {

      echo "Records created successfully\n";

   }

   $db->close();
?>
```

When the above program is executed, it will create the given records in the COMPANY table and will display the following two lines.

```
Opened database successfully

Records created successfully
```

SELECT Operation

Following PHP program shows how to fetch and display records from the COMPANY table created in the above example.

```
<?php
   class MyDB extends SQLite3

   {

      function __construct()

      {

         $this->open('test.db');

      }

   }

   $db = new MyDB();

   if(!$db){

      echo $db->lastErrorMsg();

   } else {
```

```
        echo "Opened database successfully\n";
    }


    $sql =<<<EOF
        SELECT * from COMPANY;
EOF;


    $ret = $db->query($sql);
    while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
        echo "ID = ". $row['ID'] . "\n";

        echo "NAME = ". $row['NAME'] ."\n";

        echo "ADDRESS = ". $row['ADDRESS'] ."\n";

        echo "SALARY =  ".$row['SALARY'] ."\n\n";
    }
    echo "Operation done successfully\n";
    $db->close();
?>
```

When the above program is executed, it will produce the following result.

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  20000


ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000


ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000


ID = 4
NAME = Mark
ADDRESS = Rich-Mond
```

```
SALARY =  65000


Operation done successfully
```

UPDATE Operation

Following PHP code shows how to use UPDATE statement to update any record and then fetch and display the updated records from the COMPANY table.

```php
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
   $sql =<<<EOF
      UPDATE COMPANY set SALARY = 25000.00 where ID=1;
EOF;
   $ret = $db->exec($sql);
   if(!$ret){
      echo $db->lastErrorMsg();
   } else {
      echo $db->changes(), " Record updated successfully\n";
   }

   $sql =<<<EOF
      SELECT * from COMPANY;
EOF;
   $ret = $db->query($sql);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
```

```
      echo "ID = ". $row['ID'] . "\n";

      echo "NAME = ". $row['NAME'] ."\n";

      echo "ADDRESS = ". $row['ADDRESS'] ."\n";

      echo "SALARY =  ".$row['SALARY'] ."\n\n";

   }
   echo "Operation done successfully\n";

   $db->close();

?>
```

When the above program is executed, it will produce the following result.

```
Opened database successfully

1 Record updated successfully

ID = 1

NAME = Paul

ADDRESS = California

SALARY =  25000


ID = 2

NAME = Allen

ADDRESS = Texas

SALARY =  15000


ID = 3

NAME = Teddy

ADDRESS = Norway

SALARY =  20000


ID = 4

NAME = Mark

ADDRESS = Rich-Mond

SALARY =  65000


Operation done successfully
```

DELETE Operation

Following PHP code shows how to use DELETE statement to delete any record and then fetch and display the remaining records from the COMPANY table.

169

```php
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
   $sql =<<<EOF
      DELETE from COMPANY where ID=2;
EOF;
   $ret = $db->exec($sql);
   if(!$ret){
     echo $db->lastErrorMsg();
   } else {
      echo $db->changes(), " Record deleted successfully\n";
   }


   $sql =<<<EOF
      SELECT * from COMPANY;
EOF;
   $ret = $db->query($sql);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
      echo "ID = ". $row['ID'] . "\n";
      echo "NAME = ". $row['NAME'] ."\n";
      echo "ADDRESS = ". $row['ADDRESS'] ."\n";
      echo "SALARY =   ".$row['SALARY'] ."\n\n";
   }
   echo "Operation done successfully\n";
   $db->close();
```

```
?>
```

When the above program is executed, it will produce the following result.

```
Opened database successfully
1 Record deleted successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

In this chapter, you will learn how to use SQLite in Perl programs.

Installation

SQLite3 can be integrated with Perl using Perl DBI module, which is a database access module for the Perl programming language. It defines a set of methods, variables, and conventions that provide a standard database interface.

Following are simple steps to install DBI module on your Linux/UNIX machine:

```
$ wget http://search.cpan.org/CPAN/authors/id/T/TI/TIMB/DBI-1.625.tar.gz
$ tar xvfz DBI-1.625.tar.gz
$ cd DBI-1.625
$ perl Makefile.PL
$ make
$ make install
```

If you need to install SQLite driver for DBI, then it can be installed as follows:

```
$    wget    http://search.cpan.org/CPAN/authors/id/M/MS/MSERGEANT/DBD-SQLite-
1.11.tar.gz
$ tar xvfz DBD-SQLite-1.11.tar.gz
$ cd DBD-SQLite-1.11
$ perl Makefile.PL
$ make
$ make install
```

DBI Interface APIs

Following are important DBI routines, which can suffice your requirement to work with SQLite database from your Perl program. If you are looking for a more sophisticated application, then you can look into Perl DBI official documentation.

Sr. No.	API & Description
1	**DBI->connect($data_source, "", "", \%attr)** Establishes a database connection, or session, to the requested $data_source. Returns a database handle object if the connection succeeds. Datasource has the form like: **DBI:SQLite:dbname='test.db'** where SQLite is SQLite driver name and test.db is the name of SQLite database file. If the filename is given as **':memory:'**, it will create an in-memory database in RAM that lasts only for the duration of the session. If the filename is actual device file name, then it attempts to open the database file by using its value. If no file by that name exists, then a new database file by that name gets created. You keep second and third parameter as blank strings and the last parameter is to pass various attributes as shown in the following example.
2	**$dbh->do($sql)** This routine prepares and executes a single SQL statement. Returns the number of rows affected or undef on error. A return value of -1 means the number of rows is not known, not applicable, or not available. Here, $dbh is a handle returned by DBI->connect() call.
3	**$dbh->prepare($sql)** This routine prepares a statement for later execution by the database engine and returns a reference to a statement handle object.
4	**$sth->execute()** This routine performs whatever processing is necessary to execute the prepared statement. An undef is returned if an error occurs. A successful execute always returns true regardless of the number of rows affected. Here, $sth is a statement handle returned by $dbh->prepare($sql) call.
5	**$sth->fetchrow_array()** This routine fetches the next row of data and returns it as a list containing the field values. Null fields are returned as undef values in the list.

6	**$DBI::err** This is equivalent to $h->err, where $h is any of the handle types like $dbh, $sth, or $drh. This returns native database engine error code from the last driver method called.
7	**$DBI::errstr** This is equivalent to $h->errstr, where $h is any of the handle types like $dbh, $sth, or $drh. This returns the native database engine error message from the last DBI method called.
8	**$dbh->disconnect()** This routine closes a database connection previously opened by a call to DBI->connect().

Connect to Database

Following Perl code shows how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned.

```perl
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;


print "Opened database successfully\n";
```

Now, let's run the above program to create our database test.db in the current directory. You can change your path as per your requirement. Keep the above code in sqlite.pl file and execute it as shown below. If the database is successfully created, then it will display the following message:

```
$ chmod +x sqlite.pl
$ ./sqlite.pl
Open database successfully
```

Create a Table

Following Perl program is used to create a table in the previously created database.

```perl
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                     or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(CREATE TABLE COMPANY
      (ID INT PRIMARY KEY     NOT NULL,
       NAME            TEXT    NOT NULL,
       AGE             INT     NOT NULL,
       ADDRESS         CHAR(50),
       SALARY          REAL););
my $rv = $dbh->do($stmt);
if($rv < 0){
   print $DBI::errstr;
} else {
   print "Table created successfully\n";
}
$dbh->disconnect();
```

tutorialspoint
SIMPLYEASYLEARNING

When the above program is executed, it will create COMPANY table in your test.db and it will display the following messages:

```
Opened database successfully

Table created successfully
```

Note: In case you see the following error in any of the operation:

```
DBD::SQLite::st execute failed: not an error(21) at dbdimp.c line 398
```

In such case, open **dbdimp.c file** available in DBD-SQLite installation and find out **sqlite3_prepare()** function and change its third argument to **-1** instead of **0**. Finally, install DBD::SQLite using **make** and do **make install** to resolve the problem.

INSERT Operation

Following Perl program shows how to create records in the COMPANY table created in the above example.

```perl
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (1, 'Paul', 32, 'California', 20000.00 ));
my $rv = $dbh->do($stmt) or die $DBI::errstr;

$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 ));
$rv = $dbh->do($stmt) or die $DBI::errstr;

$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
```

```
         VALUES (3, 'Teddy', 23, 'Norway', 20000.00 ));
$rv = $dbh->do($stmt) or die $DBI::errstr;


$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
         VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 ););
$rv = $dbh->do($stmt) or die $DBI::errstr;


print "Records created successfully\n";
$dbh->disconnect();
```

When the above program is executed, it will create the given records in the COMPANY table and it will display the following two lines:

```
Opened database successfully
Records created successfully
```

SELECT Operation

Following Perl program shows how to fetch and display records from the COMPANY table created in the above example.

```perl
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;
print "Opened database successfully\n";


my $stmt = qq(SELECT id, name, address, salary  from COMPANY;);
my $sth = $dbh->prepare( $stmt );
my $rv = $sth->execute() or die $DBI::errstr;
if($rv < 0){
   print $DBI::errstr;
```

```
}
while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
print "Operation done successfully\n";
$dbh->disconnect();
```

When the above program is executed, it will produce the following result.

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  20000


ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000


ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000


ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000


Operation done successfully
```

tutorialspoint
SIMPLYEASYLEARNING

UPDATE Operation

Following Perl code shows how to use UPDATE statement to update any record and then fetch and display the updated records from the COMPANY table.

```perl
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(UPDATE COMPANY set SALARY = 25000.00 where ID=1;);
my $rv = $dbh->do($stmt) or die $DBI::errstr;
if( $rv < 0 ){
   print $DBI::errstr;
}else{
   print "Total number of rows updated : $rv\n";
}
$stmt = qq(SELECT id, name, address, salary  from COMPANY;);
my $sth = $dbh->prepare( $stmt );
$rv = $sth->execute() or die $DBI::errstr;
if($rv < 0){
   print $DBI::errstr;
}
while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
```

```
print "Operation done successfully\n";

$dbh->disconnect();
```

When the above program is executed, it will produce the following result.

```
Opened database successfully

Total number of rows updated : 1

ID = 1

NAME = Paul

ADDRESS = California

SALARY =  25000


ID = 2

NAME = Allen

ADDRESS = Texas

SALARY =  15000


ID = 3

NAME = Teddy

ADDRESS = Norway

SALARY =  20000


ID = 4

NAME = Mark

ADDRESS = Rich-Mond

SALARY =  65000


Operation done successfully
```

DELETE Operation

Following Perl code shows how to use DELETE statement to delete any record and then fetch and display the remaining records from the COMPANY table.

```
#!/usr/bin/perl


use DBI;

use strict;
```

```
my $driver  = "SQLite";

my $database = "test.db";

my $dsn = "DBI:$driver:dbname=$database";

my $userid = "";

my $password = "";

my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                     or die $DBI::errstr;

print "Opened database successfully\n";


my $stmt = qq(DELETE from COMPANY where ID=2;);

my $rv = $dbh->do($stmt) or die $DBI::errstr;

if( $rv < 0 ){

   print $DBI::errstr;

}else{

   print "Total number of rows deleted : $rv\n";

}

$stmt = qq(SELECT id, name, address, salary  from COMPANY;);

my $sth = $dbh->prepare( $stmt );

$rv = $sth->execute() or die $DBI::errstr;

if($rv < 0){

   print $DBI::errstr;

}

while(my @row = $sth->fetchrow_array()) {

     print "ID = ". $row[0] . "\n";

     print "NAME = ". $row[1] ."\n";

     print "ADDRESS = ". $row[2] ."\n";

     print "SALARY =  ". $row[3] ."\n\n";

}

print "Operation done successfully\n";

$dbh->disconnect();
```

tutorialspoint
SIMPLYEASYLEARNING

When the above program is executed, it will produce the following result.

```
Opened database successfully
Total number of rows deleted : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

In this chapter, you will learn how to use SQLite in Python programs.

Installation

SQLite3 can be integrated with Python using sqlite3 module, which was written by Gerhard Haring. It provides an SQL interface compliant with the DB-API 2.0 specification described by PEP 249. You do not need to install this module separately because it is shipped by default along with Python version 2.5.x onwards.

To use sqlite3 module, you must first create a connection object that represents the database and then optionally you can create a cursor object, which will help you in executing all the SQL statements.

Python sqlite3 module APIs

Following are important sqlite3 module routines, which can suffice your requirement to work with SQLite database from your Python program. If you are looking for a more sophisticated application, then you can look into Python sqlite3 module's official documentation.

Sr. No.	API & Description
1	**sqlite3.connect(database [,timeout ,other optional arguments])** This API opens a connection to the SQLite database file. You can use **":memory:"** to open a database connection to a database that resides in RAM instead of on disk. If database is opened successfully, it returns a connection object. When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The timeout parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds). If the given database name does not exist, then this call will create the database. You can specify filename with the required path as well if you want to create a database anywhere else except in the current directory.

2	**connection.cursor([cursorClass])** This routine creates a **cursor** which will be used throughout of your database programming with Python. This method accepts a single optional parameter cursorClass. If supplied, this must be a custom cursor class that extends sqlite3.Cursor.
3	**cursor.execute(sql [, optional parameters])** This routine executes an SQL statement. The SQL statement may be parameterized (i.e., placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks and named placeholders (named style). For example:cursor.execute("insert into people values (?, ?)", (who, age))
4	**connection.execute(sql [, optional parameters])** This routine is a shortcut of the above execute method provided by the cursor object and it creates an intermediate cursor object by calling the cursor method, then calls the cursor's execute method with the parameters given.
5	**cursor.executemany(sql, seq_of_parameters)** This routine executes an SQL command against all parameter sequences or mappings found in the sequence sql.
6	**connection.executemany(sql[, parameters])** This routine is a shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor.s executemany method with the parameters given.
7	**cursor.executescript(sql_script)** This routine executes multiple SQL statements at once provided in the form of script. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter. All the SQL statements should be separated by a semicolon (;).

8	**connection.executescript(sql_script)** This routine is a shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's executescript method with the parameters given.
9	**connection.total_changes()** This routine returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.
10	**connection.commit()** This method commits the current transaction. If you don't call this method, anything you did since the last call to commit() is not visible from other database connections.
11	**connection.rollback()** This method rolls back any changes to the database since the last call to commit().
12	**connection.close()** This method closes the database connection. Note that this does not automatically call commit(). If you just close your database connection without calling commit() first, your changes will be lost!
13	**cursor.fetchone()** This method fetches the next row of a query result set, returning a single sequence, or None when no more data is available.
14	**cursor.fetchmany([size=cursor.arraysize])** This routine fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available. The method tries to fetch as many rows as indicated by the size parameter.
15	**cursor.fetchall()** This routine fetches all (remaining) rows of a query result, returning a list. An empty list is returned when no rows are available.

Connect to Database

Following Python code shows how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')

print "Opened database successfully";
```

Here, you can also supply database name as the special name **:memory:** to create a database in RAM. Now, let's run the above program to create our database **test.db** in the current directory. You can change your path as per your requirement. Keep the above code in **sqlite.py file** and execute it as shown below. If the database is successfully created, then it will display the following message.

```
$chmod +x sqlite.py
$./sqlite.py
Open database successfully
```

Create a Table

Following Python program will be used to create a table in the previously created database.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute('''CREATE TABLE COMPANY
       (ID INT PRIMARY KEY     NOT NULL,
       NAME            TEXT    NOT NULL,
       AGE             INT     NOT NULL,
       ADDRESS         CHAR(50),
       SALARY          REAL);''')
print "Table created successfully";
conn.close()
```

When the above program is executed, it will create the COMPANY table in your **test.db** and it will display the following messages:

```
Opened database successfully

Table created successfully
```

INSERT Operation

Following Python program shows how to create records in the COMPANY table created in the above example.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (1, 'Paul', 32, 'California', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");

conn.commit()
print "Records created successfully";
conn.close()
```

When the above program is executed, it will create the given records in the COMPANY table and it will display the following two lines:

```
Opened database successfully

Records created successfully
```

SELECT Operation

Following Python program shows how to fetch and display records from the COMPANY table created in the above example.

```python
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

cursor = conn.execute("SELECT id, name, address, salary  from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

When the above program is executed, it will produce the following result.

```
Opened database successfully
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  20000.0


ID =  2
NAME =  Allen
ADDRESS =  Texas
SALARY =  15000.0


ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0
```

```
ID =  4

NAME =  Mark

ADDRESS =  Rich-Mond

SALARY =  65000.0


Operation done successfully
```

UPDATE Operation

Following Python code shows how to use UPDATE statement to update any record and then fetch and display the updated records from the COMPANY table.

```
#!/usr/bin/python


import sqlite3


conn = sqlite3.connect('test.db')

print "Opened database successfully";


conn.execute("UPDATE COMPANY set SALARY = 25000.00 where ID=1")

conn.commit

print "Total number of rows updated :", conn.total_changes


cursor = conn.execute("SELECT id, name, address, salary  from COMPANY")

for row in cursor:

    print "ID = ", row[0]

    print "NAME = ", row[1]

    print "ADDRESS = ", row[2]

    print "SALARY = ", row[3], "\n"


print "Operation done successfully";

conn.close()
```

When the above program is executed, it will produce the following result.

```
Opened database successfully

Total number of rows updated : 1

ID =  1

NAME =  Paul
```

tutorialspoint
SIMPLYEASYLEARNING

```
ADDRESS =  California

SALARY =  25000.0


ID =  2

NAME =  Allen

ADDRESS =  Texas

SALARY =  15000.0


ID =  3

NAME =  Teddy

ADDRESS =  Norway

SALARY =  20000.0


ID =  4

NAME =  Mark

ADDRESS =  Rich-Mond

SALARY =  65000.0


Operation done successfully
```

DELETE Operation

Following Python code shows how to use DELETE statement to delete any record and then fetch and display the remaining records from the COMPANY table.

```python
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("DELETE from COMPANY where ID=2;")
conn.commit
print "Total number of rows deleted :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary  from COMPANY")
for row in cursor:
   print "ID = ", row[0]
```

```
    print "NAME = ", row[1]
    print "ADDRESS = ", row[2]
    print "SALARY = ", row[3], "\n"


print "Operation done successfully";
conn.close()
```

When the above program is executed, it will produce the following result.

```
Opened database successfully
Total number of rows deleted : 1
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  20000.0


ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0


ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0


Operation done successfully
```