Unit-1: Introduction to SQLite:

1.1 SQLite advantages, features and Fundamentals:

What is SQLite?

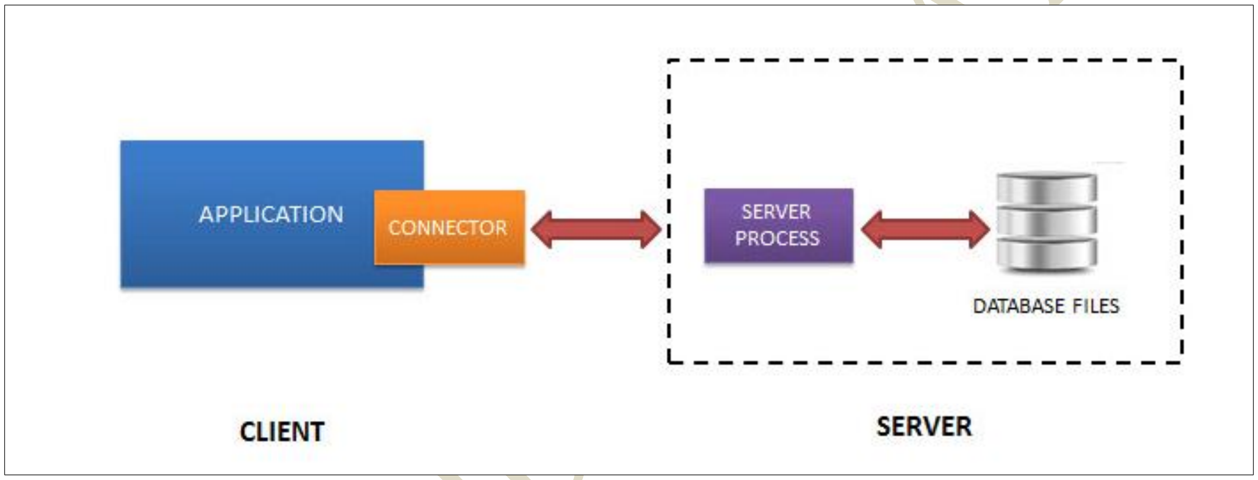
SQLite is a software library that provides a relational database management system. The lite in SQLite means lightweight in terms of setup, database administration, and required resources.

SQLite has the following noticeable features: self-contained, serverless, zero-configuration, transactional.

Serverless:

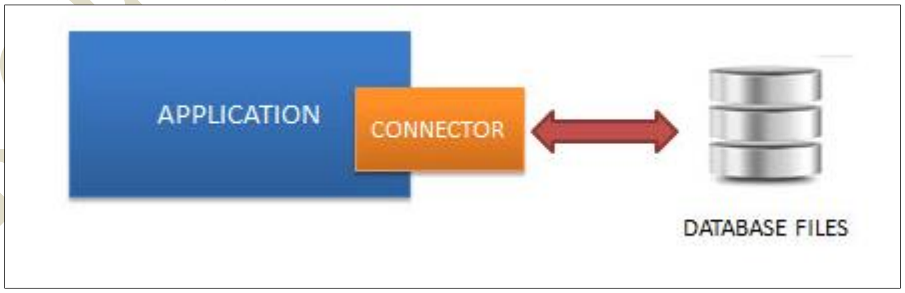
Normally, an RDBMS such as MySQL, PostgreSQL, etc., requires a separate server process to operate. The applications that want to access the database server use TCP/IP protocol to send and receive requests. This is called client/server architecture.

The following diagram illustrates the RDBMS client/server architecture:



SQLite does NOT work this way. SQLite does NOT require a server to run.
SQLite database is integrated with the application that accesses the database. The applications interact with the SQLite database read and write directly from the database files stored on disk.

The following diagram illustrates the SQLite server-less architecture:



Self-Contained:

SQLite is self-contained means it requires minimal support from the operating system or external library. This makes SQLite usable in any environment especially in embedded devices like iPhones, Android phones, game consoles, handheld media players, etc.

SQLite is developed using ANSI-C. The source code is available as a big sqlite3.c and its header file sqlite3.h. If you want to develop an application that uses SQLite, you just need to drop these files into your project and compile it with your code.

Zero-configuration:
Because of the serverless architecture, you don't need to "install" SQLite before using it. There is no server process that needs to be configured, started, and stopped. In addition, SQLite does not use any configuration files.

Transactional:
All transactions in SQLite are fully ACID-compliant. It means all queries and changes are Atomic, Consistent, Isolated, and Durable. In other words, all changes within a transaction take place completely or not at all even when an unexpected situation like application crash, power failure, or operating system crash occurs.

SQLite History
SQLite was designed originally on August 2000. It is named SQLite because it is very light weight (less than 500Kb size) unlike other database management systems like SQL Server or Oracle.

Year	Happenings
2000	SQLite was designed by D. Richard Hipp 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 Advantages
SQLite is a very popular database which has been successfully used with on disk file format for desktop applications like version control systems, financial analysis tools, media cataloging and editing suites, CAD packages, record keeping programs etc.
There are a lot of advantages to use SQLite as an application file format:

1) Lightweight
- ✓ SQLite is a very light weighted database so, it is easy to use it as an embedded software with devices like televisions, Mobile phones, cameras, home electronic devices, etc.

2) Better Performance
- ✓ Reading and writing operations are very fast for SQLite database. It is almost 35% faster than File system.
- ✓ It only loads the data which is needed, rather than reading the entire file and hold it in memory.
- ✓ If you edit small parts, it only overwrite the parts of the file which was changed.

3) No Installation Needed
- ✓ SQLite is very easy to learn. You do not need to install and configure it. Just download SQLite libraries in your computer and it is ready for creating the database.

4) Reliable

- ✓ It updates your content continuously so, little or no work is lost in a case of power failure or crash.
- ✓ SQLite is less bugs prone rather than custom written file I/O codes.
- ✓ SQLite queries are smaller than equivalent procedural codes so, chances of bugs are minimal.

5) Portable

- ✓ SQLite is portable across all 32-bit and 64-bit operating systems and big- and little-endian architectures.
- ✓ Multiple processes can be attached with same application file and can read and write without interfering each other.
- ✓ It can be used with all programming languages without any compatibility issue.

6) Accessible

- ✓ SQLite database is accessible through a wide variety of third-party tools.
- ✓ SQLite database's content is more likely to be recoverable if it has been lost. Data lives longer than code.

7) Reduce Cost and Complexity

- ✓ It reduces application cost because content can be accessed and updated using concise SQL queries instead of lengthy and error-prone procedural queries.
- ✓ SQLite can be easily extended in in future releases just by adding new tables and/or columns. It also preserve the backwards compatibility.

<u>SQLite Disadvantages</u>

- ✓ SQLite is used to handle low to medium traffic HTTP requests.
- ✓ Database size is restricted to 2GB in most cases.

SQLite Features/ Why to use SQLite

Following is a list of features which makes SQLite popular among other lightweight databases:

1. **SQLite is totally free:** SQLite is open-source. So, no license is required to work with it.
2. **SQLite is server less:** SQLite doesn't require a different server process or system to operate.
3. **SQLite is very flexible:** It facilitates you to work on multiple databases on the same session on the same time.
4. **Configuration Not Required:** SQLite doesn't require configuration. No setup or administration required.
5. **SQLite is a cross-platform DBMS:** You don't need a large range of different platforms like Windows, Mac OS, Linux, and Unix. It can also be used on a lot of embedded operating systems like Symbian, and Windows CE.
6. **Storing data is easy:** SQLite provides an efficient way to store data.
7. **Variable length of columns:** The length of the columns is variable and is not fixed. It facilitates you to allocate only the space a field needs. For example, if you have a varchar(200) column, and you put a 10 characters' length value on it, then SQLite will allocate only 20 characters' space for that value not the whole 200 space.
8. **Provide large number of API's:** SQLite provides API for a large range of programming languages. **For example:** .Net languages (Visual Basic, C#), PHP, Java, Objective C, Python and a lot of other programming language.
9. **SQLite** is written in **ANSI-C** and provides simple and easy-to-use API.

10. **SQLite** is available on UNIX (Linux, Mac OS-X, Android, iOS) and Windows (Win32, WinCE, WinRT).

1.1.1 SQLite datatype(Dynamic type, SQLite manifest typing & type affinity) (NULL, INTEGER, REAL,TEXT, BLOB)

- If you use other database systems such as MySQL and PostgreSQL, you notice that they use static typing. It means when you declare a column with a specific data type, that column can store only data of the declared data type.
- Different from other database systems, SQLite uses dynamic type system. In other words, a value stored in a column determines its data type, not the column's data type.
- In addition, you don't have to declare a specific data type for a column when you create a table. In case you declare a column with the integer data type, you can store any kind of data types such as text and BLOB, SQLite will not complain about this.
- SQLite provides five primitive data types which are referred to as **storage classes**.
- Storage classes describe the formats that SQLite uses to store data on disk. A storage class is more general than a data type e.g., INTEGER storage class includes 6 different types of integers. In most cases, you can use storage classes and data types interchangeably.

The following table illustrates 5 storage classes in SQLite:

Storage Class	**Meaning**
NULL	NULL values mean missing information or unknown.
INTEGER	Integer values are whole numbers (either positive or negative). An integer can have variable sizes such as 1, 2,3, 4, or 8 bytes.
REAL	Real values are real numbers with decimal values that use 8-byte floats.
TEXT	TEXT is used to store character data. The maximum length of TEXT is unlimited. SQLite supports various character encodings.
BLOB	BLOB stands for a **binary large object** that can store any kind of data. The maximum size of BLOB is, theoretically, unlimited.

SQLite Afinity Types

SQLite supports type affinity for columns. Any column can still store any type of data but the preferred storage class for a column is called its affinity.
There are following type affinity used to assign in SQLite3 database.

Affinity	**Description**
TEXT	This column is used to store all data using storage classes NULL, TEXT or BLOB.
NUMERIC	This column may contain values using all five storage classes.
INTEGER	It behaves the same as a column with numeric affinity with an exception in a cast expression.
REAL	It 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 don't persuade data from one storage class into another.

SQLite Affinity and Type Names

Following is a list of various data types names which can be used while creating SQLite tables.

Data Types	**Corresponding Affinity**
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER
CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB	TEXT
BLOB no datatype specified	NONE
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC

Date and Time Data Type

In SQLite, there is no separate class to store dates and times. But you can store date and times as TEXT, REAL or INTEGER values.

Storage Class	**Date Format**
TEXT	It specifies a date in a format like "yyyy-mm-dd hh:mm:ss.sss".
REAL	It specifies the number of days since noon in Greenwich on November 24, 4714 B.C.
INTEGER	It specifies the number of seconds since 1970-01-01 00:00:00 utc.

Boolean Data Type

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

1.1.2 Transaction, Rollback, Commit

SQLite & ACID

SQLite is a transactional database that all changes and queries are atomic, consistent, isolated, and durable(ACID).

SQLite guarantees all the transactions are ACID compliant even if the transaction is interrupted by a program crash, operation system dump, or power failure to the computer.

- **Atomic:** a transaction should be atomic. It means that a change cannot be broken down into smaller ones. When you commit a transaction, either the entire transaction is applied or not.
- **Consistent:** a transaction must ensure to change the database from one valid state to another. When a transaction starts and executes a statement to modify data, the database becomes inconsistent. However, when the transaction is committed or rolled back, it is important that the transaction must keep the database consistent.
- **Isolation:** a pending transaction performed by a session must be isolated from other sessions. When a session starts a transaction and executes the INSERT or UPDATE statement to change the data, these changes are only visible to the current session, not others. On the other hand, the changes committed by other sessions after the transaction started should not be visible to the current session.
- **Durable:** if a transaction is successfully committed, the changes must be permanent in the database regardless of the condition such as power failure or program crash. On the contrary, if the program crashes before the transaction is committed, the change should not persist.

SQLite transaction statements
By default, SQLite operates in auto-commit mode. It means that for each command, SQLite starts, processes, and commits the transaction automatically.
To start a transaction explicitly, you use the following steps:
First, open a transaction by issuing the BEGIN TRANSACTION command.

```
BEGIN TRANSACTION;
```

After executing the statement BEGIN TRANSACTION, the transaction is open until it is explicitly committed or rolled back.
Second, issue SQL statements to select or update data in the database. Note that the change is only visible to the current session (or client).
Third, commit the changes to the database by using the COMMIT or COMMIT TRANSACTION statement.

```
COMMIT;
```

If you do not want to save the changes, you can roll back using the ROLLBACK or ROLLBACK TRANSACTION statement:

```
ROLLBACK;
```

SQLite transaction example
We will create two new tables: accounts and account_changes for the demonstration. The accounts table stores data about the account numbers and their balances. The account_changes table stores the changes of the accounts.

First, create the accounts and account_changes tables by using the following CREATE TABLE statements:

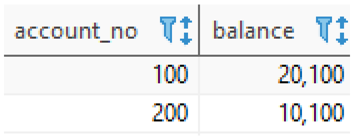
```
CREATE TABLE accounts (
        account_no INTEGER NOT NULL,
        balance DECIMAL NOT NULL DEFAULT 0,
        PRIMARY KEY(account_no),
      CHECK(balance >= 0)
);
CREATE TABLE account_changes (
        change_no INT NOT NULL PRIMARY KEY,
        account_no INTEGER NOT NULL,
        flag TEXT NOT NULL,
        amount DECIMAL NOT NULL,
        changed_at TEXT NOT NULL
);
```

Second, insert some sample data into the accounts table.

```
INSERT INTO accounts (account_no,balance)
VALUES (100,20100);

INSERT INTO accounts (account_no,balance)
VALUES (200,10100);
```

Third, query data from the accounts table:

```
SELECT * FROM accounts;
```



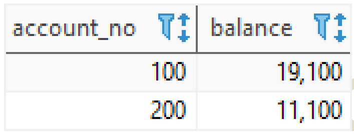
account_no	balance
100	20,100
200	10,100

Fourth, transfer 1000 from account 100 to 200, and log the changes to the table account_changes in a single transaction.

```
BEGIN TRANSACTION;

UPDATE accounts
   SET balance = balance - 1000
 WHERE account_no = 100;

UPDATE accounts
   SET balance = balance + 1000
 WHERE account_no = 200;

INSERT INTO account_changes(account_no,flag,amount,changed_at)
VALUES(100,'-',1000,datetime('now'));

INSERT INTO account_changes(account_no,flag,amount,changed_at)
VALUES(200,'+',1000,datetime('now'));

COMMIT;
```

Fifth, query data from the accounts table:

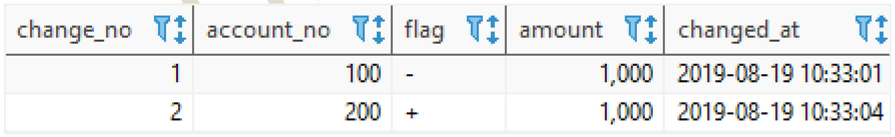
```
SELECT * FROM accounts;
```



account_no	balance
100	19,100
200	11,100

As you can see, balances have been updated successfully.
Sixth, query the contents of the account_changes table:

```
SELECT * FROM account_changes;
```



change_no	account_no	flag	amount	changed_at
1	100	-	1,000	2019-08-19 10:33:01
2	200	+	1,000	2019-08-19 10:33:04

Let's take another example of rolling back a transaction.
First, attempt to deduct 20,000 from account 100:

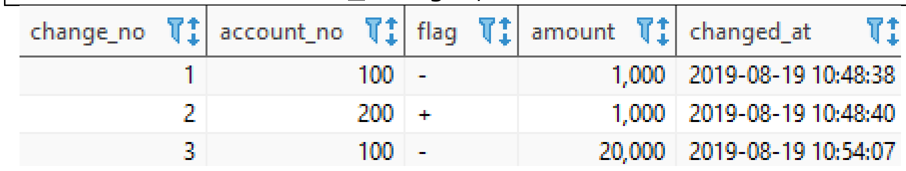
```
BEGIN TRANSACTION;

UPDATE accounts
   SET balance = balance - 20000
 WHERE account_no = 100;

INSERT INTO account_changes(account_no,flag,amount,changed_at)
VALUES(100,'-',20000,datetime('now'));
```

SQLite issued an error due to not enough balance:

```
[SQLITE_CONSTRAINT]  Abort due to constraint violation (CHECK constraint failed:
accounts)
```

However, the log has been saved to the account_changes table:

```
SELECT * FROM account_changes;
```



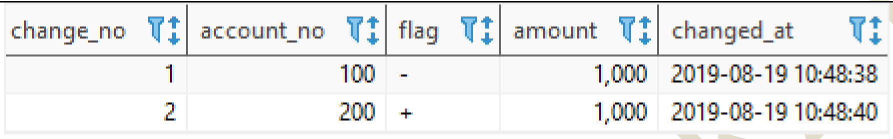
change_no	account_no	flag	amount	changed_at
1	100	-	1,000	2019-08-19 10:48:38
2	200	+	1,000	2019-08-19 10:48:40
3	100	-	20,000	2019-08-19 10:54:07

Second, roll back the transaction by using the ROLLBACK statement:

```
ROLLBACK;
```

Finally, query data from the account_changes table, you will see that the change no #3 is not there anymore:

```
SELECT * FROM account_changes;
```



change_no	account_no	flag	amount	changed_at
1	100	-	1,000	2019-08-19 10:48:38
2	200	+	1,000	2019-08-19 10:48:40

1.2 Data Filtering and Triggers

SQLite Logical Operator

Following is a list of logical operators in SQLite:

Operator	Description
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
NOT IN	It is the negation of IN operator which is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
GLOB	The GLOB operator is used to compare a value to similar values using wildcard operators. Also, glob is case sensitive, unlike like.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. For example: EXISTS, NOT BETWEEN, NOT IN, etc. These are known as 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 NOT operator work like !=
||	This operator is used to add two different strings and make new one.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

1.2.1 Filtering: Distinct, where, between, in, like, Union, intersect, Except, limit, IS NULL

SQLite DISTINCT Clause

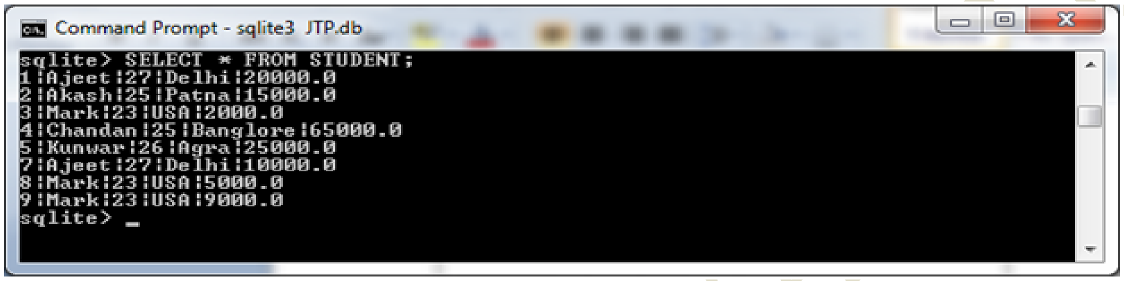
The SQLite DISTINCT clause is used with SELECT statement to eliminate all the duplicate records and fetching only unique records.
It is used when you have multiple duplicate records in the table.

Syntax:

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

Example:

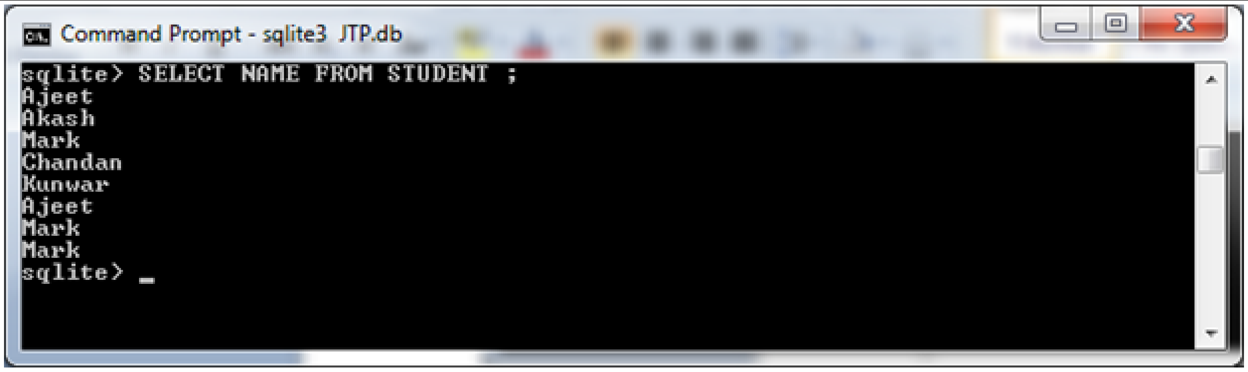
We have a table named "STUDENT", having the following data:



```
sqlite> SELECT * FROM STUDENT;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
3|Mark|23|USA|2000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
7|Ajeet|27|Delhi|10000.0
8|Mark|23|USA|5000.0
9|Mark|23|USA|9000.0
sqlite> _
```

First Select NAME from "STUDENT" without using DISTINCT keyword. It will show the duplicate records:

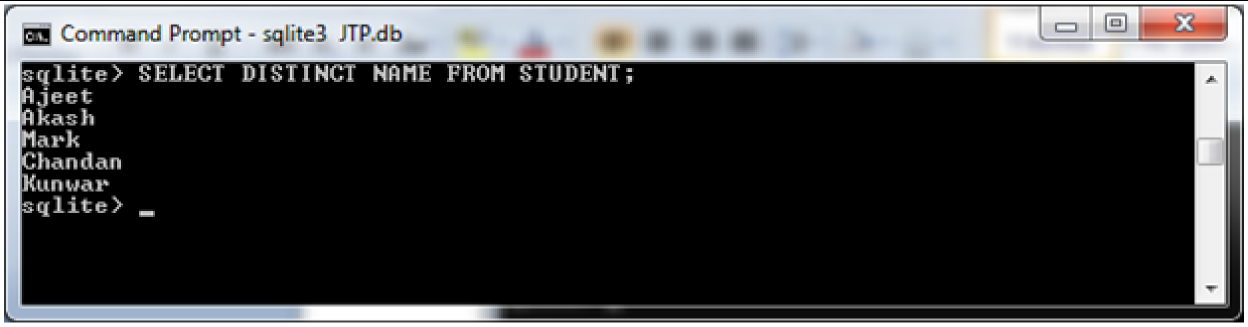
```
SELECT NAME FROM STUDENT;
```



```
sqlite> SELECT NAME FROM STUDENT ;
Ajeet
Akash
Mark
Chandan
Kunwar
Ajeet
Mark
Mark
sqlite> _
```

Now, select NAME from "STUDENT" using DISTINCT keyword.

```
SELECT DISTINCT NAME FROM STUDENT;
```



```
sqlite> SELECT DISTINCT NAME FROM STUDENT;
Ajeet
Akash
Mark
Chandan
Kunwar
sqlite> _
```

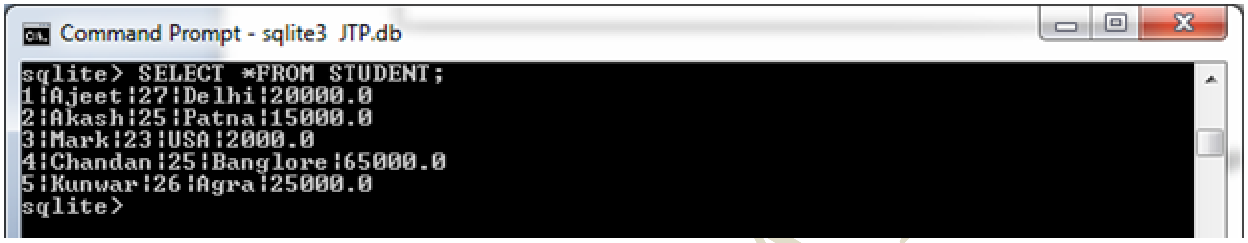
SQLite WHERE Clause

The SQLite WHERE clause is generally used with SELECT, UPDATE and DELETE statement to specify a condition while you fetch the data from one table or multiple tables.

If the condition is satisfied or true, it returns specific value from the table. You would use WHERE clause to filter the records and fetching only necessary records.
WHERE clause is also used to filter the records and fetch only specific data.
Syntax:

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

Example:
In this example, we are using WHERE clause with several comparison and logical operators. like >, <, =, LIKE, NOT, etc.
We have a table student having the following data:



```
sqlite> SELECT *FROM STUDENT;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
3|Mark|23|USA|2000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
sqlite>
```

Example1:
Select students WHERE age is greater than or equal to 25 and fees is greater than or equal to 10000.00

```
SELECT * FROM STUDENT WHERE AGE >= 25 AND FEES >= 10000.00;
```

Example2:
Select students form STUDENT table where name starts with 'A' doesn't matter what come after 'A'.

```
SELECT * FROM STUDENT WHERE NAME LIKE 'A%';
```

Example3:
Select all students from STUDENT table where age is either 25 or 27.

```
SELECT * FROM STUDENT WHERE AGE IN ( 25, 27 );
```

Example4:
Select all students from STUDENT table where age is neither 25 nor 27.

```
SELECT * FROM STUDENT WHERE AGE NOT IN (25,27);
```

SQLite LIKE Clause (Operator)
The SQLite LIKE operator is used to match text values against a pattern using wildcards. In the case search expression is 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 (%)
- The underscore (_)

The percent sign represents zero, one, or multiple numbers or characters. The underscore represents a single number or character.
Syntax:

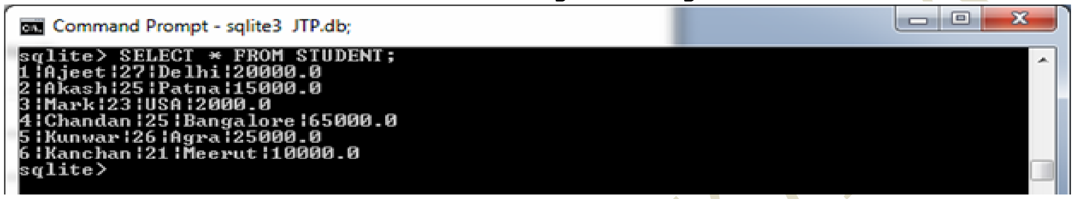
```
SELECT FROM table_name
WHERE column LIKE 'XXXX%';
```

```
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_';
```

Here, XXXX could be any numeric or string value.

Example:

We have a table named 'STUDENT' having following data:



Command Prompt - sqlite3 JTP.db;

```
sqlite> SELECT * FROM STUDENT;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
3|Mark|23|USA|2000.0
4|Chandan|25|Bangalore|65000.0
5|Kunwar|26|Agra|25000.0
6|Kanchan|21|Meerut|10000.0
sqlite>
```

In these examples the **WHERE** statement having different LIKE clause with '%' and '_' operators and operation is done on 'FEES':

Statement	**Description**
Where FEES like '200%'	It will find any values that start with 200.
Where FEES like '%200%'	It will find any values that have 200 in any position.
Where FEES like '_00%'	It will find any values that have 00 in the second and third positions.
Where FEES like '2_%_%'	It will find any values that start with 2 and are at least 3 characters in length.
Where FEES like '%2'	It will find any values that end with 2
Where FEES like '_2%3'	It will find any values that have a 2 in the second position and end with a 3
Where FEES like '2___3'	It will find any values in a five-digit number that start with 2 and end with 3

Example1: Select all records from STUDENT table WHERE age start with 2.

```
SELECT * FROM STUDENT WHERE AGE  LIKE '2%';
```

Example2:

Select all records from the STUDENT table WHERE ADDRESS will have "a" (a) inside the text:

```
SELECT * FROM STUDENT WHERE ADDRESS LIKE '%a%';
```

SQLite Union Operator

SQLite UNION Operator is used to combine the result set of two or more tables using SELECT statement. The UNION operator shows only the unique rows and removes duplicate rows.

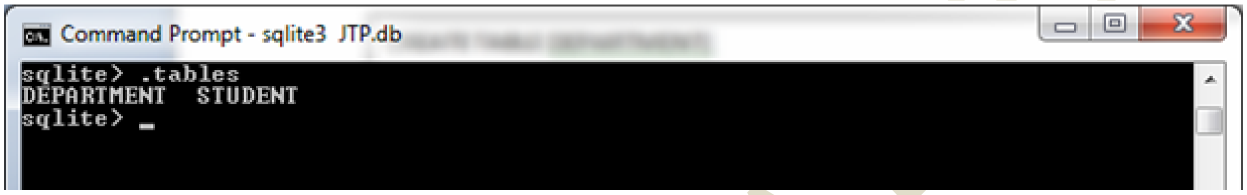
While using UNION operator, each SELECT statement must have the same number of fields in the result set.

Syntax:

```
SELECT expression1, expression2, ... expression_n
FROM tables
[WHERE conditions]
UNION
SELECT expression1, expression2, ... expression_n
FROM tables
[WHERE conditions];
```

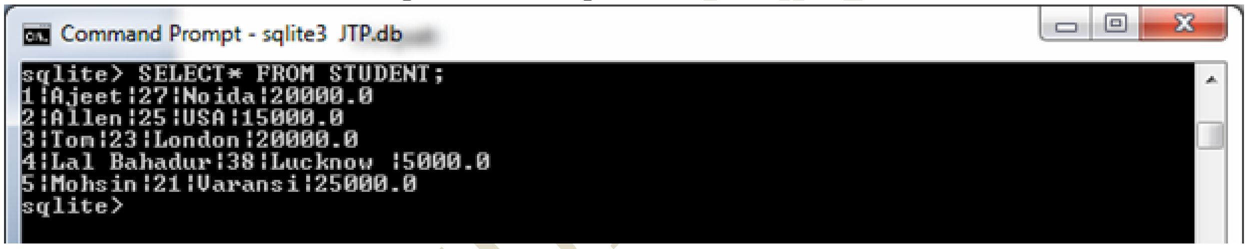
Example:

We have two tables "STUDENT" and "DEPARTMENT".



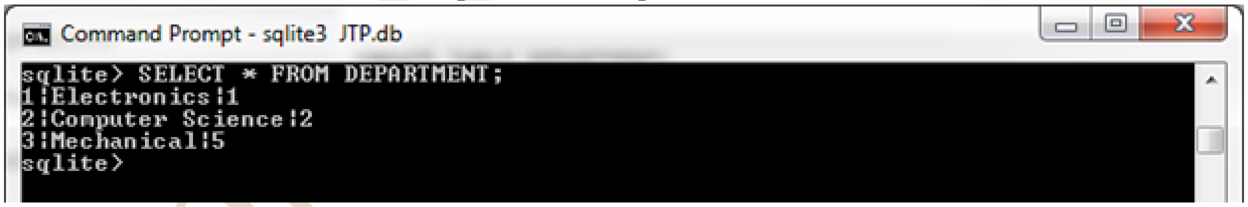
```
Command Prompt - sqlite3 JTP.db
sqlite> .tables
DEPARTMENT   STUDENT
sqlite> _
```

The "STUDENT" table is having the following data:



```
Command Prompt - sqlite3 JTP.db
sqlite> SELECT* FROM STUDENT;
1|Ajeet|27|Noida|20000.0
2|Allen|25|USA|15000.0
3|Tom|23|London|20000.0
4|Lal Bahadur|38|Lucknow |5000.0
5|Mohsin|21|Varansi|25000.0
sqlite>
```

The "DEPARTMENT" table is having the following data:



```
Command Prompt - sqlite3 JTP.db
sqlite> SELECT * FROM DEPARTMENT;
1|Electronics|1
2|Computer Science|2
3|Mechanical|5
sqlite>
```

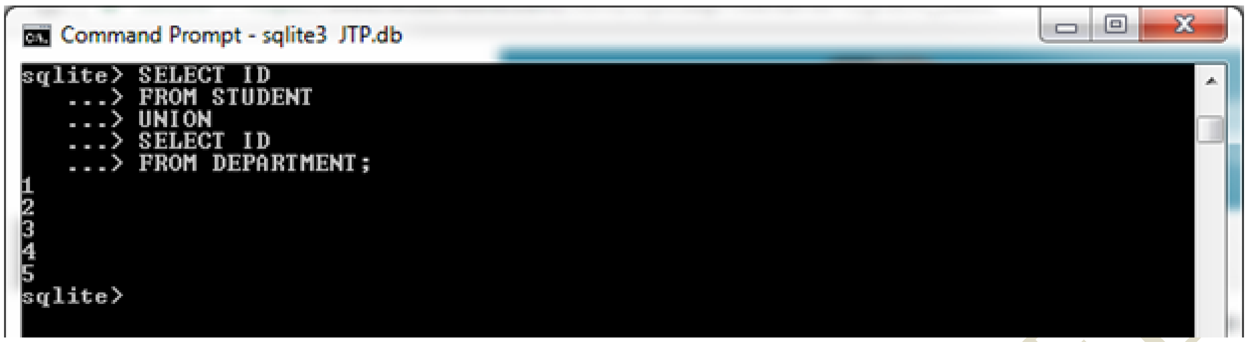
Example1: Return Single Field

This simple example returns only one field from multiple SELECT statements where the both fields have same data type.

Let's take the above two tables "STUDENT" and "DEPARTMENT" and select id from both table to make UNION.

```
SELECT ID FROM STUDENT
UNION
SELECT ID FROM DEPARTMENT;
```

Output:


Command Prompt - sqlite3 JTP.db

```
sqlite> SELECT ID
   ...> FROM STUDENT
   ...> UNION
   ...> SELECT ID
   ...> FROM DEPARTMENT;
1
2
3
4
5
sqlite>
```

SQLite INTERSECT operator
SQLite INTERSECT operator compares the result sets of two queries and returns distinct rows that are output by both queries.
The following illustrates the syntax of the INTERSECT operator:

```
SELECT select_list1
FROM table1
INTERSECT
SELECT select_list2
FROM table2
```

The basic rules for combining the result sets of two queries are as follows:

- First, the number and the order of the columns in all queries must be the same.
- Second, the data types must be comparable.

For the demonstration, we will create two tables t1 and t2 and insert some data into both:

```
CREATE TABLE t1(
   v1 INT
);
INSERT INTO t1(v1)
VALUES(1),(2),(3);

CREATE TABLE t2(
   v2 INT
);
INSERT INTO t2(v2)
VALUES(2),(3),(4);
```

The following statement illustrates how to use the INTERSECT operator to compare result sets of two queries:

```
SELECT v1
FROM t1
INTERSECT
SELECT v2
FROM t2;
```

OUTPUT:

v1
2
3

SQLite EXCEPT operator

SQLite EXCEPT operator compares the result sets of two queries and returns distinct rows from the left query that are not output by the right query.

The following shows the syntax of the EXCEPT operator:

```
SELECT select_list1
FROM table1
EXCEPT
SELECT select_list2
FROM table2
```

This query must conform to the following rules:

- First, the number of columns in the select lists of both queries must be the same.
- Second, the order of the columns and their types must be comparable.

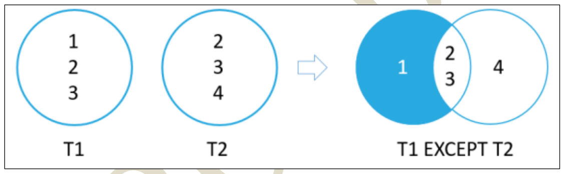
The following statements create two tables t1 and t2 and insert some data into both tables:

```
CREATE TABLE t1( v1 INT);
INSERT INTO t1(v1) VALUES(1),(2),(3);
CREATE TABLE t2(v2 INT );
INSERT INTO t2(v2) VALUES(2),(3),(4);
```

The following statement illustrates how to use the EXCEPT operator to compare result sets of two queries:

```
SELECT v1
FROM t1
EXCEPT
SELECT v2
FROM t2;
```

The output is 1.



SQLite LIMIT clause

The LIMIT clause is an optional part of the SELECT statement. You use the LIMIT clause to constrain the number of rows returned by the query.

For example, a SELECT statement may return one million rows. However, if you just need the first 10 rows in the result set, you can add the LIMIT clause to the SELECT statement to retrieve 10 rows.

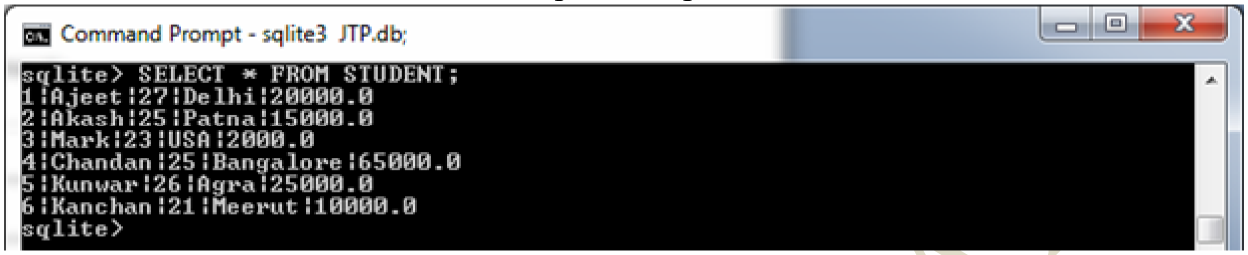
The following illustrates the syntax of the LIMIT clause.

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows]
```

The LIMIT clause can also be used along with OFFSET clause.

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows] OFFSET [row num]
```

Example:
We have a table named 'STUDENT' having following data:

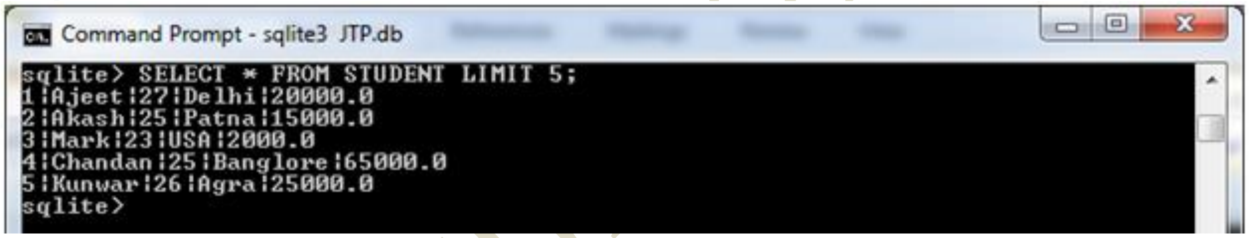

Command Prompt - sqlite3 JTP.db;

```
sqlite> SELECT * FROM STUDENT;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
3|Mark|23|USA|2000.0
4|Chandan|25|Bangalore|65000.0
5|Kunwar|26|Agra|25000.0
6|Kanchan|21|Meerut|10000.0
sqlite>
```

Example1:
Fetch the records from the "STUDENT" table by using LIMIT according to your need of number of rows.

```
SELECT * FROM STUDENT LIMIT 5;
```

Output:

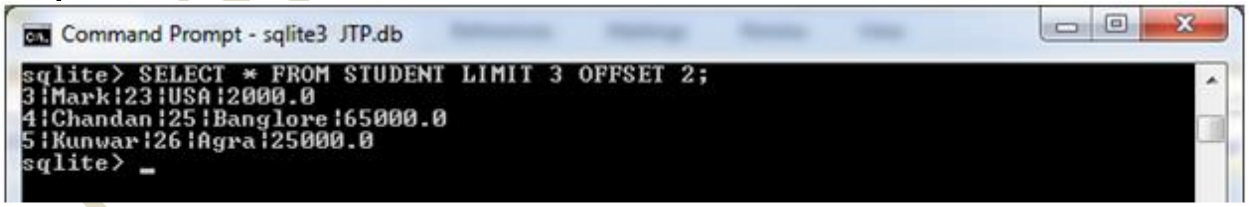

Command Prompt - sqlite3 JTP.db

```
sqlite> SELECT * FROM STUDENT LIMIT 5;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
3|Mark|23|USA|2000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
sqlite>
```

Example2:
OFFSET is used to not retrieve the offset records from the table. It is used in some cases where we have to retrieve the records starting from a certain point:
Select 3 records form table "STUDENT" starting from 3rd position(start with 0).

```
SELECT * FROM STUDENT LIMIT 3 OFFSET 2;
```

Output:


Command Prompt - sqlite3 JTP.db

```
sqlite> SELECT * FROM STUDENT LIMIT 3 OFFSET 2;
3|Mark|23|USA|2000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
sqlite> _
```

SQLite IS NULL
The SQLite IS NULL Condition is used to test for a NULL value in a SELECT, INSERT, UPDATE, or DELETE statement.
Syntax:

```
expression IS NULL
```

Note
- If expression is a NULL value, the condition evaluates to TRUE.
- If expression is not a NULL value, the condition evaluates to FALSE.

Example - With SELECT Statement
Let's look at an example of how to use SQLite IS NULL in a SELECT statement:

```
SELECT *
FROM employees
WHERE department IS NULL;
```

This SQLite IS NULL example will return all records from the employees table where the department contains a NULL value.

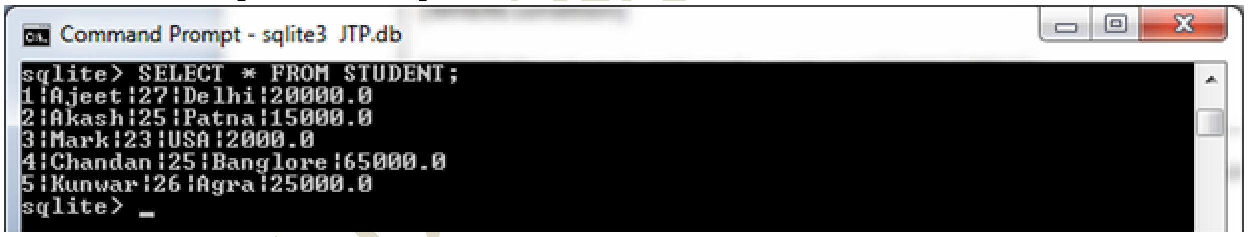
1.2.2 Having, Group by, Order by, Conditional Logic (CASE)

SQLite ORDER BY Clause
The SQLite ORDER BY clause is used to sort the fetched data in ascending or descending order, based on one or more column.
Syntax:

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

You can use one or more columns in ORDER BY clause. Your used column must be presented in column-list.

Let's take an example to demonstrate ORDER BY clause. We have a table named "STUDENT" having the following data:

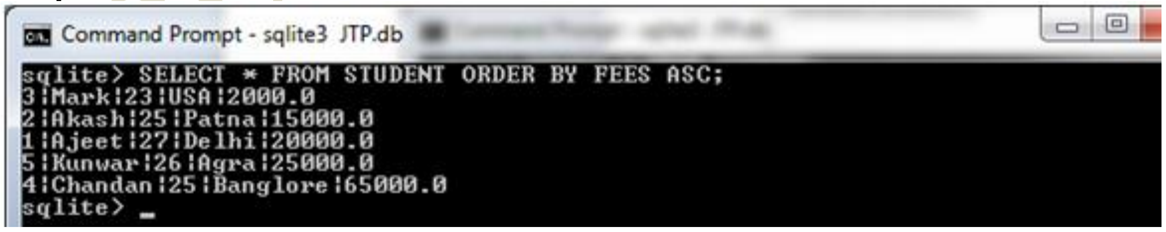

Command Prompt - sqlite3 JTP.db

```
sqlite> SELECT * FROM STUDENT;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
3|Mark|23|USA|2000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
sqlite> _
```

Example1:
Select all records from "STUDENT" where FEES is in ascending order:

```
SELECT * FROM STUDENT ORDER BY FEES ASC;
```

Output:

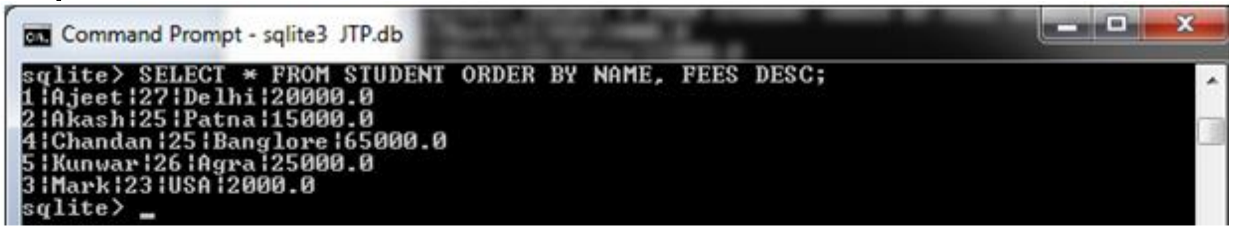

Command Prompt - sqlite3 JTP.db

```
sqlite> SELECT * FROM STUDENT ORDER BY FEES ASC;
3|Mark|23|USA|2000.0
2|Akash|25|Patna|15000.0
1|Ajeet|27|Delhi|20000.0
5|Kunwar|26|Agra|25000.0
4|Chandan|25|Banglore|65000.0
sqlite> _
```

Example2:
Fetch all data from the table "STUDENT" and sort the result in descending order by NAME and FEES:

```
SELECT * FROM STUDENT ORDER BY NAME, FEES DESC;
```

Output:



Command Prompt - sqlite3 JTP.db

```
sqlite> SELECT * FROM STUDENT ORDER BY NAME, FEES DESC;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
3|Mark|23|USA|2000.0
sqlite> _
```

SQLite GROUP BY Clause

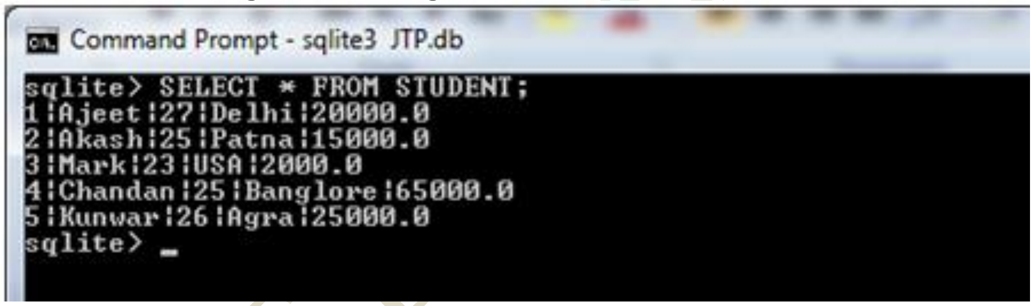
The SQLite GROUP BY clause is used with SELECT statement to collaborate the same identical elements into groups.

The GROUP BY clause is used with WHERE clause in SELECT statement and precedes the ORDER BY clause.

Syntax:

```
SELECT column-list
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2....columnN
ORDER BY column1, column2....columnN
```

Let's take an example to demonstrate the GROUP BY clause. We have a table named "STUDENT", having the following data:



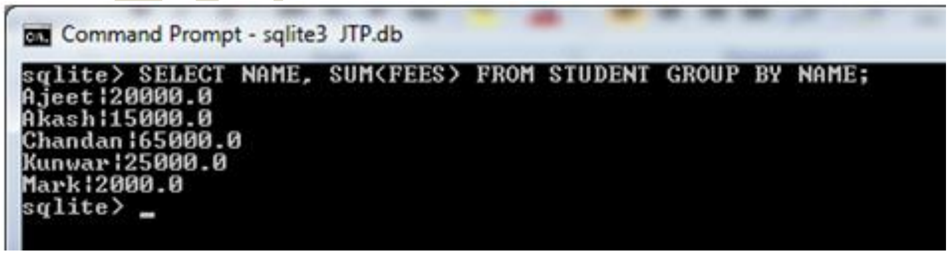
Command Prompt - sqlite3 JTP.db

```
sqlite> SELECT * FROM STUDENT;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
3|Mark|23|USA|2000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
sqlite> _
```

Use the GROUP BY query to know the total amount of FEES of each student:

```
SELECT NAME, SUM(FEES) FROM STUDENT GROUP BY NAME;
```

Output:

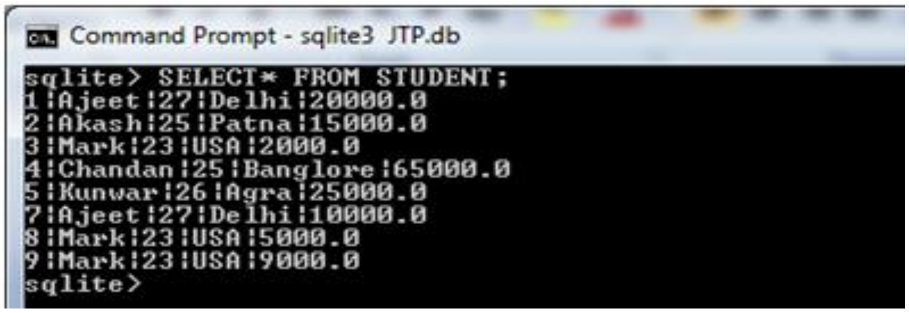


Command Prompt - sqlite3 JTP.db

```
sqlite> SELECT NAME, SUM(FEES) FROM STUDENT GROUP BY NAME;
Ajeet|20000.0
Akash|15000.0
Chandan|65000.0
Kunwar|25000.0
Mark|2000.0
sqlite> _
```

Now, create some more records in "STUDENT" table using the following INSERT statement:

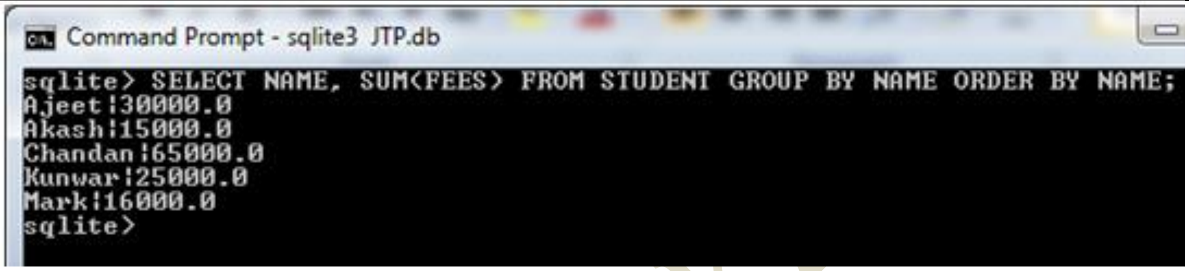
```
INSERT INTO STUDENT VALUES (7, 'Ajeet', 27, 'Delhi', 10000.00 );
INSERT INTO STUDENT VALUES (8, 'Mark', 23, 'USA', 5000.00 );
INSERT INTO STUDENT VALUES (9, 'Mark', 23, 'USA', 9000.00 );
```



```
Command Prompt - sqlite3  JTP.db
sqlite> SELECT* FROM STUDENT;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
3|Mark|23|USA|2000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
7|Ajeet|27|Delhi|10000.0
8|Mark|23|USA|5000.0
9|Mark|23|USA|9000.0
sqlite>
```

The new updated table has the inserted entries. Now, use the same GROUP BY statement to group-by all the records using NAME column:

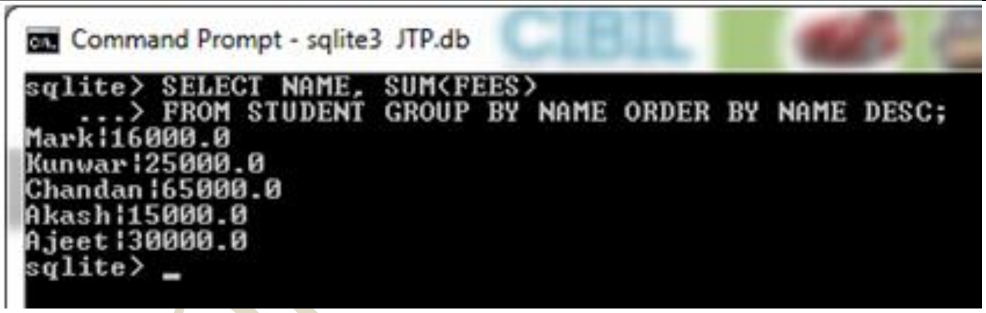
```
SELECT NAME, SUM(FEES) FROM STUDENT GROUP BY NAME ORDER BY NAME;
```



```
Command Prompt - sqlite3  JTP.db
sqlite> SELECT NAME, SUM(FEES) FROM STUDENT GROUP BY NAME ORDER BY NAME;
Ajeet|30000.0
Akash|15000.0
Chandan|65000.0
Kunwar|25000.0
Mark|16000.0
sqlite>
```

You can use ORDER BY clause along with GROUP BY to arrange the data in ascending or descending order.

```
SELECT NAME, SUM(FEES)
FROM STUDENT GROUP BY NAME ORDER BY NAME DESC;
```



```
Command Prompt - sqlite3  JTP.db
sqlite> SELECT NAME, SUM(FEES)
   ...> FROM STUDENT GROUP BY NAME ORDER BY NAME DESC;
Mark|16000.0
Kunwar|25000.0
Chandan|65000.0
Akash|15000.0
Ajeet|30000.0
sqlite> _
```

SQLite HAVING Clause
The SQLite HAVING clause is used to specify conditions that filter which group results appear in the final results. The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

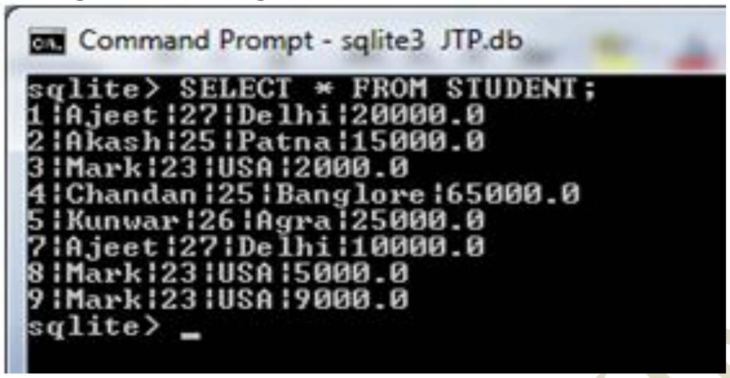
The position of HAVING clause in a SELECT query:

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

Syntax:

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
ORDER BY column1, column2
```

Example:
Let's take an example to demonstrate HAVING Clause. We have a table named "STUDENT", having the following data:

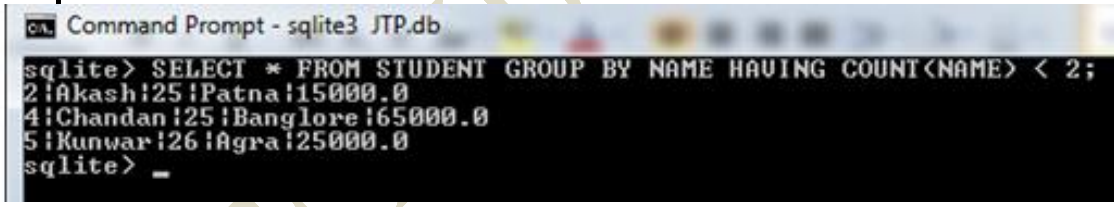


```
sqlite> SELECT * FROM STUDENT;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
3|Mark|23|USA|2000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
7|Ajeet|27|Delhi|10000.0
8|Mark|23|USA|5000.0
9|Mark|23|USA|9000.0
sqlite> _
```

Example1:
Display all records where name count is less than 2:

```
SELECT * FROM STUDENT GROUP BY NAME HAVING COUNT(NAME) < 2;
```

Output:

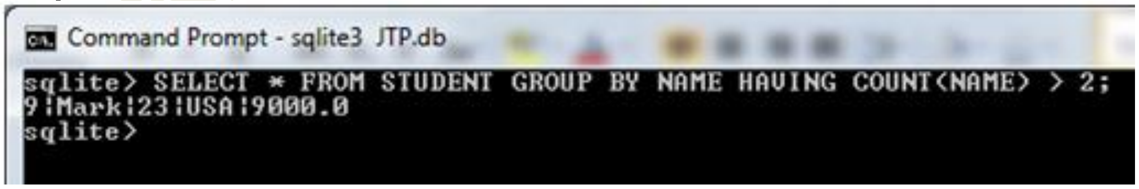


```
sqlite> SELECT * FROM STUDENT GROUP BY NAME HAVING COUNT(NAME) < 2;
2|Akash|25|Patna|15000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
sqlite> _
```

Example2:
Display all records where name count is greater than 2:

```
SELECT * FROM STUDENT GROUP BY NAME HAVING COUNT(NAME) > 2;
```

Output:



```
sqlite> SELECT * FROM STUDENT GROUP BY NAME HAVING COUNT(NAME) > 2;
9|Mark|23|USA|9000.0
sqlite>
```

SQLite Conditional Logic (CASE)Statement

- In SQLite CASE statement is like an if...then...else condition in other programming languages or like C Language switch statements.
- Generally, the SQLite Case statement contains an optional expression followed by one or more WHEN ... THEN clauses and it is finished with an optional ELSE clause and a required END keyword.
- In SQLite we can use Case statement with Select, Update, Delete, Where, Order By, Having clauses to get required values by defining multiple conditions based on our requirement.

Syntax of SQLite Case Statement

```
CASE test_expression

WHEN [condition.1] THEN [expression.1]

WHEN [condition.2] THEN [expression.2]

...

WHEN [condition.n] THEN [expression.n]

ELSE [expression]

END
```

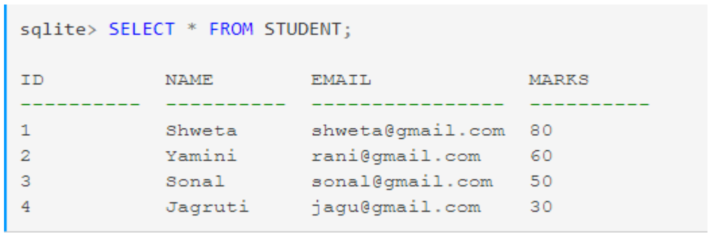
Here in SQLite Case statement each WHEN. .. THEN clauses evaluated in an orderly manner. First, it evaluated condition 1 in case if it satisfied then it returns expression 1 otherwise it will execute condition 2 and so on. If no condition is satisfied, then finally execution goes to ELSE block, and expression under ELSE is evaluated.

Example of SQLite Case Statement

We will see how to use the SQLite case statement with Select query for that create one table called STUDENT and insert some data by using the following queries.

```
CREATE TABLE STUDENT
(ID INTEGER PRIMARY KEY,
NAME TEXT NOT NULL,
EMAIL TEXT,
MARKS FLOAT);

INSERT INTO STUDENT
values (1,'Shweta','shweta@gmail.com',80),
(2,'Yamini','rani@gmail.com', 60),
(3,'Sonal','sonal@gmail.com', 50),
(4,'Jagruti','jagu@gmail.com', 30);
```

Once we create and insert data in the STUDENT table execute the following query to check records in the table.

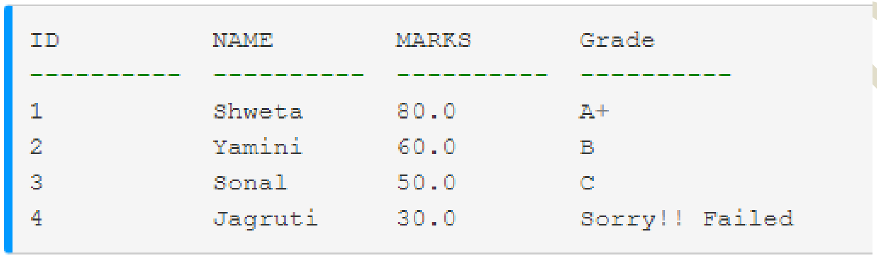


```
sqlite> SELECT * FROM STUDENT;

ID          NAME        EMAIL             MARKS
----------  ----------  ----------------  ----------
1           Shweta      shweta@gmail.com  80
2           Yamini      rani@gmail.com    60
3           Sonal       sonal@gmail.com   50
4           Jagruti     jagu@gmail.com    30
```

Following is the example of using the SQLite Case Statement with Select query.

```
SELECT ID, NAME, MARKS,
CASE
        WHEN MARKS >=80 THEN 'A+'
        WHEN MARKS >=70 THEN 'A'
        WHEN MARKS >=60 THEN 'B'
        WHEN MARKS >=50 THEN 'C'

ELSE 'Sorry!! Failed'

END as 'Grade'

FROM STUDENT;
```

Output:



```
ID          NAME        MARKS       Grade
----------  ----------  ----------  ----------
1           Shweta      80.0        A+
2           Yamini      60.0        B
3           Sonal       50.0        C
4           Jagruti     30.0        Sorry!! Failed
```

1.3 SQLite joins: Inner, left, cross, self, Full outer joins.

SQLite Joins
In SQLite, joins are used to get result set by combining or joining two or more tables. By using the JOIN clause, we can join multiple tables together and get records based on our requirements.

Suppose in SQLite we have two tables (Employees, Departments) and both tables is having common column called DepartmentId and if we want to get Employee details who are mapped to department then by using Joins in SQLite we can easily get it.

In SQLite, Joins are the best way to combine and get records from multiple tables. Generally, in SQLite each JOIN operator will combine two tables into another table. In case if we have three or more tables then those tables can be joined by using multiple JOIN operators in SQLite statements.

In SQLite, we have different types of JOINS available those are

1. Inner Join
2. Outer Join
3. Cross Join
4. Self Join

Inner Join - In SQLite inner joins are used to return only matching records from tables based on the defined conditions.

Outer Join - In SQLite outer joins are used to return records from tables even if defined conditions not satisfying. Generally, we have three types of outer joins available those are Left Outer Joins, Right Outer Joins and Full Outer Joins but SQLite will support only Left Outer Joins.

Cross Join - In SQLite cross joins are used to get the Cartesian product of tables.

Self Join - In SQLite self-join is used to join the same table.

1. SQLite Inner Join

In SQLite, INNER JOIN is used to combine and return only matching records from multiples tables based on the conditions defined in SQLite statements.

Syntax of SQLite Inner Join

```
SELECT t1.col1, t2.col2, ...
FROM table1 t1 [INNER] JOIN table2 t2
ON t1.col3=t2.col5;
```

Example : We need to create two tables dept_master and emp_master and need to insert some data for that use following queries.

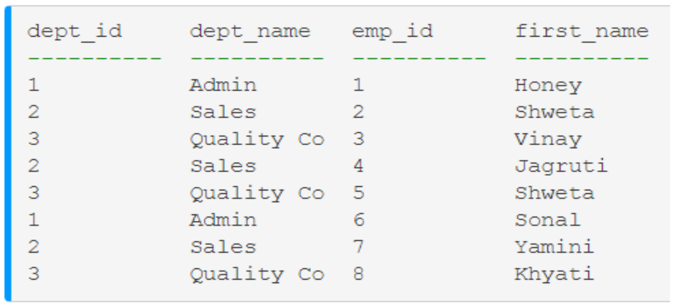
```
CREATE TABLE dept_master
(dept_id INTEGER PRIMARY KEY AUTOINCREMENT,
dept_name TEXT);
INSERT INTO dept_master(dept_name)
VALUES('Admin'),
('Sales'),
('Quality Control'),
('Marketing');

CREATE TABLE emp_master
(emp_id INTEGER PRIMARY KEY AUTOINCREMENT,
first_name TEXT,
last_name TEXT,
salary NUMERIC,
dept_id INTEGER);

INSERT INTO emp_master
values (1,'Honey','Patel',10100,1),
(2,'Shweta','Jariwala', 19300,2),
(3,'Vinay','Jariwala', 35100,3),
(4,'Jagruti','Viras', 9500,2),
(5,'Shweta','Rana',12000,3),
(6,'sonal','Menpara', 13000,1),
(7,'Yamini','Patel', 10000,2),
(8,'Khyati','Shah', 500000,3);
```

SQLite query like as shown following to use INNER JOIN with a select statement.

```
SELECT d.dept_id, dept_name, emp_id, first_name
FROM dept_master d JOIN emp_master e
ON d.dept_id = e.dept_id;
```

If you observe above SQLite INNER JOIN query we are joining dept_master and emp_master tables and applied Inner Join on dept_id column to get only employees whose dep_id equals with dept_id in department table.

Following is the result of SQLite INNER JOIN example.



```
dept_id     dept_name   emp_id      first_name
----------  ----------  ----------  ----------
1           Admin       1           Honey
2           Sales       2           Shweta
3           Quality Co  3           Vinay
2           Sales       4           Jagruti
3           Quality Co  5           Shweta
1           Admin       6           Sonal
2           Sales       7           Yamini
3           Quality Co  8           Khyati
```

Here in above example, there is no employee from Marketing department that's the reason marketing department records are not included in the result.

2. SQLite Left Outer Join

In SQLite, by using Inner Join we can get only matching rows from multiple tables based on the conditions defined in select statements. If we use SQLite Outer Join it will select matching rows from multiple tables same as Inner Join and some other rows outside of the relationship.

In simple terms we can say SQLite OUTER JOIN is an addition of INNER JOIN. Generally we have three types of Outer Joins in SQL standard those are LEFT, RIGHT and FULL Outer Joins but SQLite supports only LEFT OUTER JOIN.

Syntax of Left Outer Join

```
SELECT t1.col1,t2.col2,...FROM
table1 t1 LEFT OUTER JOIN table2 t2
ON t1.col3=t2.col5;
```

In the above SQLite Left Outer Join, table1 is a left-hand table, and table2 is a right-hand table. Here the left outer join tries to match every row of table1 table with every row in table2 table based on the join condition (t1.col3 = t2.col5) and it returns matching rows from both the tables and remaining rows of table1 table that doesn't match with table2 table are also included in the result.

Example: Consider above tables dep_master and emp_master for Left Outer JOIN with a select statement.

```
SELECT d.dept_id,dept_name,emp_id,first_name
FROM dept_master d LEFT OUTER JOIN emp_master e
ON d.dept_id=e.dept_id;
```

Here dept_master is a left-hand table so all the rows of the dept_master table included in the result even though rows are unmatched and it included only matched rows of emp_master table.

In emp_master table, we don't have any employee for the marketing department so in result emp_id and first_name are null.

3. SQLite Cross Join

In SQLite, CROSS JOIN is used to get the Cartesian product of rows by matching each row of the first table with every row of the second table.
By using the CROSS JOIN keyword in SQLite statements we can get the result which contains a combination of all the rows from the first table with all the rows of the second table.
In SQLite Cross join resultant table will contain multiplication number of rows in input tables. Suppose if table1 contains 10 rows and table2 contains 5 rows then if we apply Cross Join on these two tables we will get a resultant table that contains 50 rows.

Syntax of SQLite Cross Join

```
SELECT * FROM table1 CROSS JOIN table2
```

Example: Consider above tables dep_master and emp_master for Cross JOIN with a select statement.

```
SELECT dept_name,emp_id,first_name
FROM dept_master CROSS JOIN emp_master;
```

If you observe above result all the rows of dept_master table matched with all the row of emp_master and returned Cartesian product of dept_master and emp_master tables.

4. SQLite Self Join

In SQLite Self Join is used to join the same table with itself. To use SQLite Self Join we need to create different alias names for the same table to perform operations based on our requirements.
Suppose if we need to compare columns in same table then Self Join will help us to achieve our functionality. For example, we have a table called Employees with three columns Employee Id, Employee Name, Manager Id and we want to get the Employees who are managers in this situation we need to join Employees table with itself.

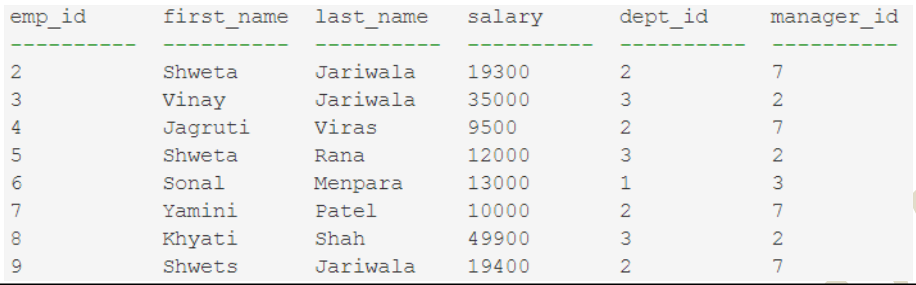
Syntax of SQLite Self Join

```
SELECT x.column_name, y.column_name...
FROM table1 x, table1 y
WHERE x.column_name1 = y.column_name1;
```

If you observe above SQLite Self join syntax we given alias name to table1 as x and y and used same field of table1 table for comparison.

Example of SQLite Self Join

```
CREATE TABLE emp_master
(emp_id INTEGER PRIMARY KEY AUTOINCREMENT,
first_name TEXT,
last_name TEXT,
salary NUMERIC,
dept_id INTEGER,
manager_id INTEGER);


INSERT INTO emp_master
values (2,'Shweta','Jariwala', 19300,2,7),
(3,'Vinay','Jariwala', 35100,3,2),
(4,'Jagruti','Viras', 9500,2,7),
(5,'Shweta','Rana',12000,3,2),
(6,'sonal','Menpara', 13000,1,3),
(7,'Yamini','Patel', 10000,2,7),
(8,'Khyati','Shah', 49900,3,2),
(9,'Shwets','Jariwala',19400,2,7);

sqlite> SELECT * FROM emp_master;
```



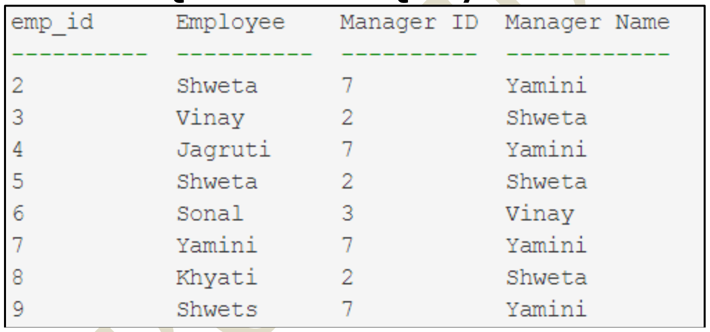
```
emp_id      first_name  last_name   salary      dept_id     manager_id
----------  ----------  ----------  ----------  ----------  ----------
2           Shweta      Jariwala    19300       2           7
3           Vinay       Jariwala    35000       3           2
4           Jagruti     Viras       9500        2           7
5           Shweta      Rana        12000       3           2
6           Sonal       Menpara     13000       1           3
7           Yamini      Patel       10000       2           7
8           Khyati      Shah        49900       3           2
9           Shwets      Jariwala    19400       2           7
```

Example : to use Self Join with a select statement to get employees manager details.

```
SELECT x.emp_id, x.first_name as Employee, y.emp_id as 'Manager
ID', y.first_name as 'Manager Name'
FROM emp_master x, emp_master y
WHERE x.manager_id = y.emp_id;
```

If you observe the above example we used the emp_master table twice with different alias names (x, y) to get a list of employees with their manager's details.

Result of SQLite Self Join Query



```
emp_id      Employee    Manager ID  Manager Name
----------  ----------  ----------  ------------
2           Shweta      7           Yamini
3           Vinay       2           Shweta
4           Jagruti     7           Yamini
5           Shweta      2           Shweta
6           Sonal       3           Vinay
7           Yamini      7           Yamini
8           Khyati      2           Shweta
9           Shwets      7           Yamini
```

1.4 SQLite Trigger:

- SQLite Trigger is an event-driven action or database callback function which is invoked automatically when an INSERT, UPDATE, and DELETE statement is performed on a specified table.
- The main tasks of triggers are like enforcing business rules, validating input data, and keeping an audit trail.

Usage of Triggers:

- Triggers are used for enforcing business rules.
- Validating input data.
- Generating a unique value for a newly-inserted row in a different file.
- Write to other files for audit trail purposes.
- Query from other files for cross-referencing purposes.
- Used to access system functions.
- Replicate data to different files to achieve data consistency.

Advantages of using triggers:

1. Triggers make the application development faster. Because the database stores triggers, you do not have to code the trigger actions into each database application.
2. Define a trigger once and you can reuse it for many applications that use the database.
3. Maintenance is easy. If the business policy changes, you have to change only the corresponding trigger program instead of each application program.

Disadvantages of using triggers:

1. It is hard to follow their logic as it they can be fired before or after the database insert/update happens.
2. It is easy to forget about triggers and if there is no documentation it will be difficult to figure out for new developers for their existence.
3. Triggers run every time when the database fields are updated and it is overhead on system. It makes system run slower.

1.4.1 Concepts of Trigger, Before and After trigger (on Insert, Update, Delete)

Every trigger associated with a table has a unique name and function based on two factors:
1. Time: BEFORE or AFTER a specific row event.
2. Event: INSERT, UPDATE or DELETE.

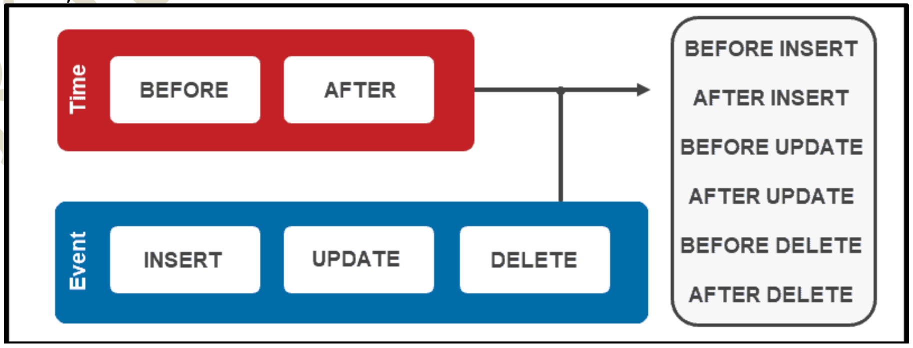


Figure: SQLite Triggers

Syntax of Creating SQLite Trigger:

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
[BEFORE|AFTER|INSTEAD OF] [INSERT|UPDATE|DELETE]
ON table_name
[FOR EACH ROW | FOR EACH STATEMENT] [WHEN condition]
BEGIN
Trigger_action_body
END;
```

If you observe the above syntax we defined multiple parameters to create trigger we will learn all the parameters in detail.
IF NOT EXISTS – It's Optional and it will prevent throwing error in case if we try to create an existing trigger.
Trigger_name – Its name of the trigger which we are going to create.
[BEFORE|AFTER|INSTEAD OF] – It will determine when the trigger action can be performed is it BEFORE or AFTER or INSTEAD OF the event.
[INSERT|UPDATE|DELETE] – It will determine in which action triggers can be invoked such as INSERT, UPDATE or DELETE.
table_name – Its name of the table on which we are going to create trigger.
[FOR EACH ROW | FOR EACH STATEMENT] – Present SQLite will support only FOR EACH ROW so we don't need to explicitly specify FOR EACH ROW and its Optional.
Trigger_action_body – It contains SQLite statements which will execute whenever the trigger action performed.

Triggers allow access to values from the table for comparison purposes using NEW and OLD. The availability of the modifiers depends on the trigger event you use:

Trigger Event	**OLD**	**NEW**
INSERT	No	Yes
UPDATE	Yes	Yes
DELETE	Yes	No

Checking or modifying a value when trying to insert data makes the NEW.<column name> modifier available. This is because a table is updated with new content. In contrast, an OLD.<column name> value does not exist for an insert statement because there is no information exists in its place beforehand.

When updating a table row, both modifiers are available. There is OLD.<colum name> data which we want to update to NEW.<column name> data.

Finally, when removing a row of data, the OLD.<column name> modifier accesses the removed value. The NEW.<column name> does not exist because nothing is replacing the old value upon removal.

Example: To use triggers with examples for that create new tabled called "Product" using the following statements.

```
CREATE TABLE Product
(pid INTEGER PRIMARY KEY,
pname TEXT NOT NULL,
amount REAL,
quantity INTEGER);
```

SQLite BEFORE INSERT Trigger

Now we will create a trigger (trg_validate_products_before_insert) on the "Product" table to raise **before insert** of any data using the following statements.

```
CREATE TRIGGER trg_validate_products_before_insert BEFORE INSERT
ON Product
BEGIN
SELECT
        CASE
        WHEN NEW.quantity < 0 THEN
        RAISE(ABORT, 'Invalid Quantity')
        END;
SELECT
        CASE
        WHEN NEW.amount<=0 THEN
        RAISE(ABORT, 'Invalid Amount')
        END;
END;
```

We used a **NEW** reference to access quantity and amount columns and added validation to check quantity and amount values while inserting a new row.

Now we will try to insert invalid amount value in Product table using following statements.

```
INSERT INTO Product(pid,pname,amount,quantity) VALUES(1,'Marbles',-5,20);

Error: Invalid Amount
```

The above statement returns Invalid Amount because we added validation to allow new row insertion only when quantity and amount column values are greater than zero.

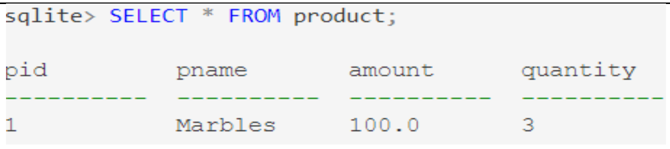
Now we will see what will happen when we try to insert invalid quantity value in the Product table using the following statements.

```
INSERT INTO Product(pid,pname,amount,quantity) VALUES(1,'Marbles',100,-3);

Error: Invalid Quantity
```

The above statement also returns Invalid Quantity because we tried to insert value which is less than zero.

Now we will try to insert valid data in the Product table using the following statements.

```
INSERT INTO Product(pid,pname,amount,quantity) VALUES(1,'Marbles',100,3);
```



```
sqlite> SELECT * FROM product;

pid         pname       amount      quantity
----------  ----------  ----------  ----------
1           Marbles     100.0       3
```

SQLite AFTER UPDATE Trigger
Now we will see how to raise triggers after updating the table records using AFTER UPDATE statement with example.
By using AFTER UPDATE we can raise the trigger immediately after completion of query execution.
To see how to raise the trigger immediately after completion of query execution first, we will create a new table called products_log using following statements.

```
CREATE TABLE Product_log
(pid INTEGER,
operation TEXT,
old_amount REAL,
new_amount REAL,
old_pname TEXT,
new_pname TEXT);
```

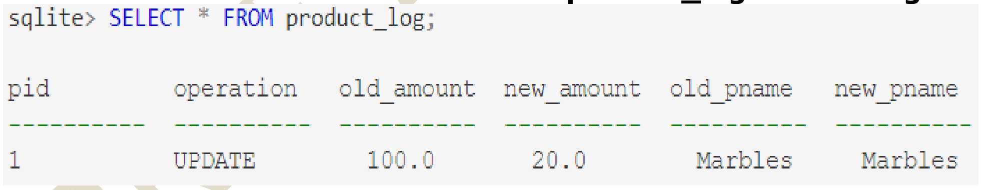
Now, create an AFTER UPDATE trigger to log the data into the Product_logs table whenever there is an update in the Product table amount or pname columns. By this, we can keep track of product prices in the future also.

```
CREATE TRIGGER trg_product_after_update AFTER UPDATE
on Product
WHEN OLD.amount <> NEW.amount
OR OLD.pname <> NEW.pname
BEGIN
INSERT INTO Product_log
(pid,operation,old_amount,new_amount,old_pname,new_pname)
VALUES(OLD.pid,'UPDATE',
OLD.amount,NEW.amount,OLD.pname,NEW.pname);
END;
```

In above trigger statement we defined condition in WHEN clause to invoke trigger only when there is a change in pname or amount columns of Product table.
Now we will update amount for pid 1 to 20rs using following statement.

```
UPDATE product SET amount = 20 WHERE pid = 1;
```

Now we will check the records of the product_log table using following statement.

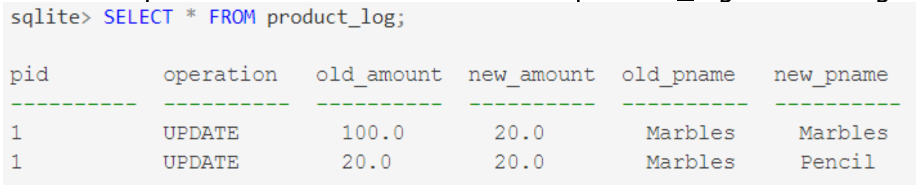


```
sqlite> SELECT * FROM product_log;

pid         operation   old_amount  new_amount  old_pname   new_pname
----------  ----------  ----------  ----------  ----------  ----------
1           UPDATE        100.0       20.0        Marbles     Marbles
```

Now, we will try to update pname to "pencil" in the Product table where pid is 1.

```
UPDATE Product SET pname ='Pencil' WHERE pid = 1;
```

Once we update let's check the records of product_log table using following statement.



```
sqlite> SELECT * FROM product_log;

pid         operation   old_amount  new_amount  old_pname   new_pname
----------  ----------  ----------  ----------  ----------  ----------
1           UPDATE        100.0       20.0        Marbles     Marbles
1           UPDATE        20.0        20.0        Marbles     Pencil
```

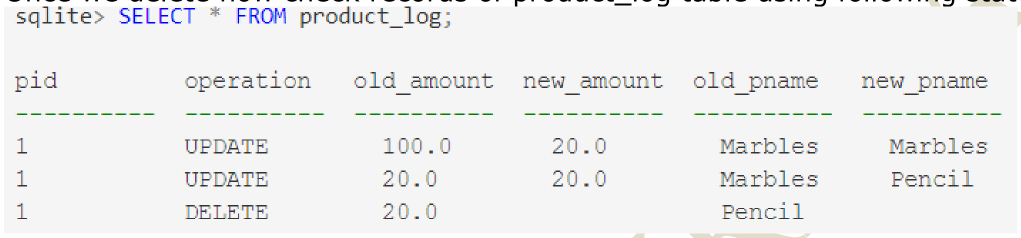
SQLite AFTER DELETE Trigger
Now we will see how to raise the trigger after deleting the records in a table using AFTER DELETE statement.
We will create a trigger to log data in the product_log table whenever we delete any records from the product table using following statements.

```
CREATE TRIGGER trg_product_after_delete AFTER DELETE on Product
BEGIN
INSERT into product_log(pid,operation,old_pname,old_amount)
VALUES(OLD.pid,'DELETE ' || date('now'),OLD.pname,OLD.amount);
END;
```

Now, let's delete one record from Product table using following query.

```
sqlite> DELETE FROM product WHERE pid=1;
```

Once we delete now check records of product_log table using following statement.



```
sqlite> SELECT * FROM product_log;

pid         operation   old_amount  new_amount  old_pname   new_pname
----------  ----------  ----------  ----------  ----------  ----------
1           UPDATE       100.0       20.0        Marbles     Marbles
1           UPDATE       20.0        20.0        Marbles     Pencil
1           DELETE       20.0                    Pencil
```

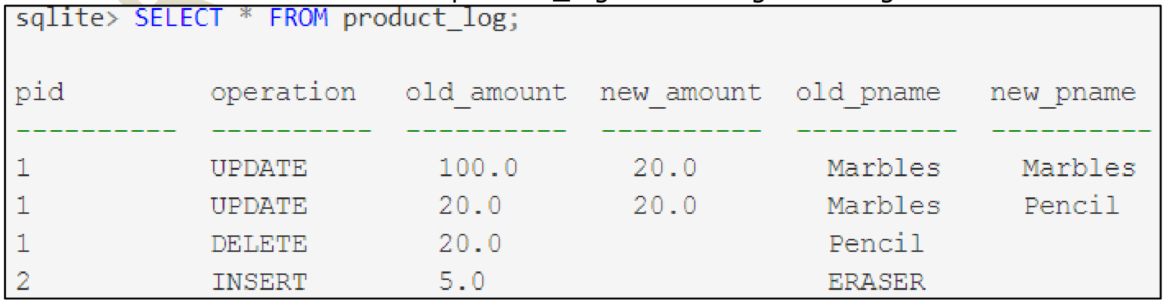
SQLite AFTER INSERT Trigger
Now we will see how to raise triggers after insertion of data in the table using AFTER INSERT statement with example.
Create a trigger using the following statements to raise the trigger and log the data in the product_log table whenever we insert new records in the Product table.

```
CREATE TRIGGER trg_product_after_insert AFTER INSERT on Product
BEGIN
INSERT into product_log
(pid,operation,old_pname,old_amount)
VALUES(NEW.pid,'INSERT ' || date('now'),NEW.pname,NEW.amount);
END;
```

Now, let's insert one record into Product table using following statement.

```
INSERT INTO product (pid,pname,quantity,amount) VALUES(2,'ERASER', 200,5);
```

Now we will check the records of product_log table using following statement.



```
sqlite> SELECT * FROM product_log;

pid         operation   old_amount  new_amount  old_pname   new_pname
----------  ----------  ----------  ----------  ----------  ----------
1           UPDATE       100.0       20.0        Marbles     Marbles
1           UPDATE       20.0        20.0        Marbles     Pencil
1           DELETE       20.0                    Pencil
2           INSERT       5.0                     ERASER
```

SQLite BEFORE UPDATE Trigger
Now we will see how to raise trigger before updating the table records using BEFORE UPDATE statement with example.
Let's create a trigger with the BEFORE UPDATE statement using the following statements on the Product table.

```
CREATE TRIGGER trg_validate_products_before_update BEFORE UPDATE
ON Product
BEGIN
SELECT
        CASE
        WHEN NEW.quantity < 0 THEN
        RAISE(ABORT,'Invalid Quantity')
        END;
SELECT
        CASE
        WHEN NEW.amount<=0 THEN
        RAISE(ABORT,'Invalid Amount')
        END;
END;
```

If you observe above statements we added validation that amount of product and quantity must be greater than zero and we used a **NEW** reference to access quantity and amount columns of row that is going to be updated in the Product table.
Now we will try to update invalid amount to Product table using following statements.

```
UPDATE Product SET Amount = -3 WHERE pid = 2;

Error: Invalid Amount
```

When we try to update invalid value in the Amount column of Product table it throws error like as shown.
Now we will try to update invalid quantity to Product table using following statement.

```
UPDATE Product SET quanity = -10 WHERE pid = 2;

Error: Invalid Quantity
```

In both cases when we try to update invalid values in amount and quantity columns we got validation errors because of our BEFORE UPDATE trigger.

SQLite BEFORE DELETE Trigger
Now we will see how to raise trigger before deleting the table records using BEFORE DELETE statement with example.
Consider we have two tables called publisher and book like as shown following.

```
CREATE TABLE Publisher(
Pub_id INTEGER PRIMARY KEY,
Pub_name TEXT NOT NULL);

CREATE TABLE Book(
Book_id INTEGER PRIMARY KEY,
Book_name TEXT NOT NULL,
Price Real,
pub_id INTEGER);

INSERT INTO Publisher (Pub_id, Pub_name)
```

```
VALUES (1,'Oreilly'),
(2,'Tata Mcgraw Hill'),
(3,'Indian Express'),
(4,'Packt');

INSERT INTO Book (Book_id, Book_name, Price, pub_id)
VALUES(1,'C++',400,2),
(2,'Javascript',600,1),
(3,'Oracle',780,2),
(4,'C',150,3),
(5,'Sqlite',900,4);

SELECT * FROM Publisher;
SELECT * from Book;
```

Here the pub_id column in book table is a foreign key references to publisher table.

Now we will create a trigger on Publisher table using BEFORE DELETE statement.

```
CREATE TRIGGER trg_publisher_before_delete BEFORE DELETE
on Publisher
BEGIN
SELECT
   CASE
      WHEN (SELECT count(pub_id) FROM book WHERE pub_id=OLD.pub_id)>0 THEN
      RAISE(ABORT,"Child table having data")
   END;
END;
```

Here if we try to delete data from the Publisher table based on pub_id then first it will check that same pub_id available in a book table or not. In case if it available then it will raise exceptions like "Child table having data" otherwise it will delete the data.

Let's try to delete record from the publisher table where pub_id is 1 using following statements.

```
sqlite>DELETE FROM publisher WHERE pub_id=1;

Error: Child table having data
```

1.4.2 Create, Drop trigger, Disable and Enable trigger

SQLite Delete / Remove Trigger

In SQLite by using the DROP TRIGGER command we can easily remove or delete triggers which are associated with tables.

Syntax to Delete Trigger

```
DROP TRIGGER [IF EXISTS] trigger_name
```

In the above syntax IF EXISTS will help us to prevent throwing errors in case if we are trying to delete triggers that do not exist in database.

Example to Delete Trigger

```
DROP TRIGGER trg_product_after_insert;
```

Note : *Note that triggers are automatically dropped when the associated table is dropped.*