

**G H Raisonni College of
Engineering & Management,
Wagholi, Pune – 412 207**

e Pune

**Department of Cyber Security and
Data Science**

**Database Design Using SQLite and
Firebase (Value Added Course)
Lab Manual
Semester-V**

(2024-25)

Faculty Name:

Prof. Sharayu Anap

Experiment No. 1

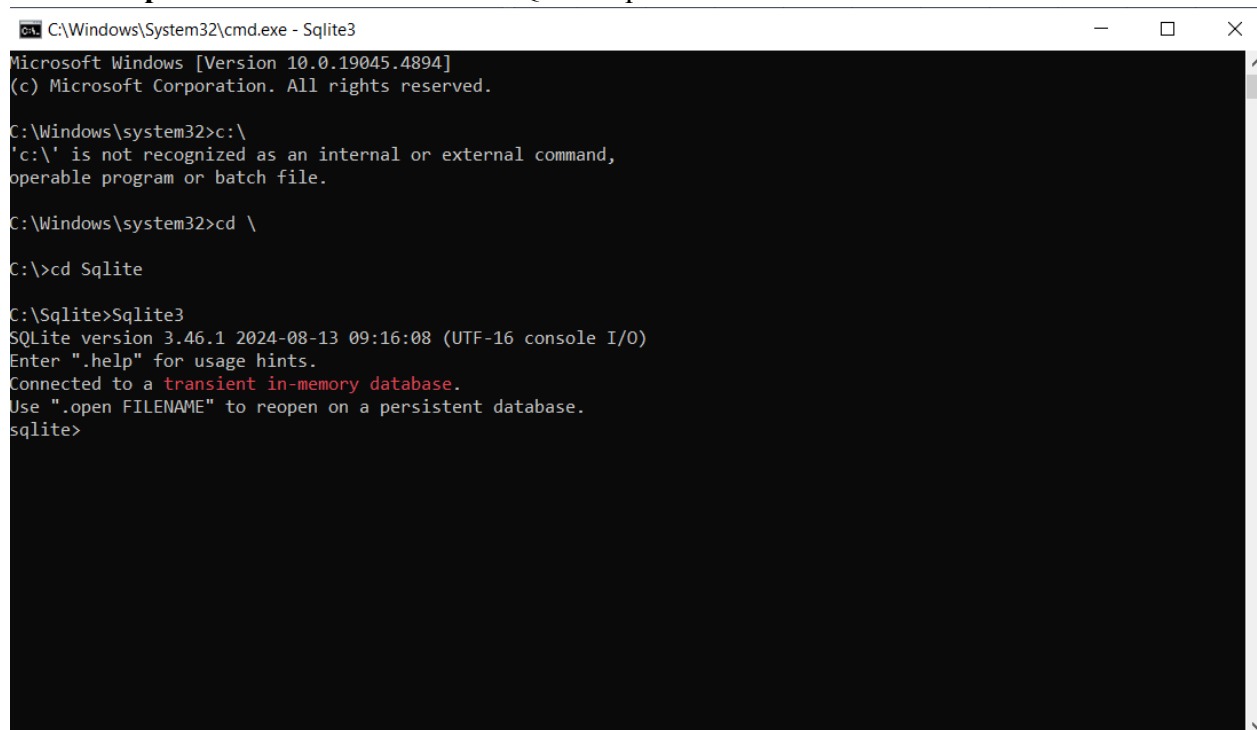
Title: Installing and Setting Up SQLite

Aim: To Install and Setting Up SQLite

Theory:

Install SQLite on Windows

- **Step 1** – Go to <https://www.sqlite.org/download.html> , and download precompiled binaries from Windows section.
- **Step 2** – Download `sqlite-dll-win-x64-3460100.zip` and `sqlite-tools-win-x64-3460100.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** – Go to the command prompt and issue `sqlite3` command, which should display the following result.
- **Step 5** – Provide command `C:\SQLite>Sqlite`



```
C:\Windows\System32\cmd.exe - Sqlite3
Microsoft Windows [Version 10.0.19045.4894]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\system32>c:\
'c:\' is not recognized as an internal or external command,
operable program or batch file.

C:\Windows\system32>cd \

C:\>cd Sqlite

C:\Sqlite>Sqlite3
SQLite version 3.46.1 2024-08-13 09:16:08 (UTF-16 console I/O)
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

Conclusion: Thus we performed installation and Setting Up SQLite

Experiment No. 2

Title: Create a simple database and table.

Aim: To Create a simple database and table.

Theory:

Syntax

Following is the basic syntax of sqlite3 command to create a database: –

`$sqlite3 DatabaseName.db`

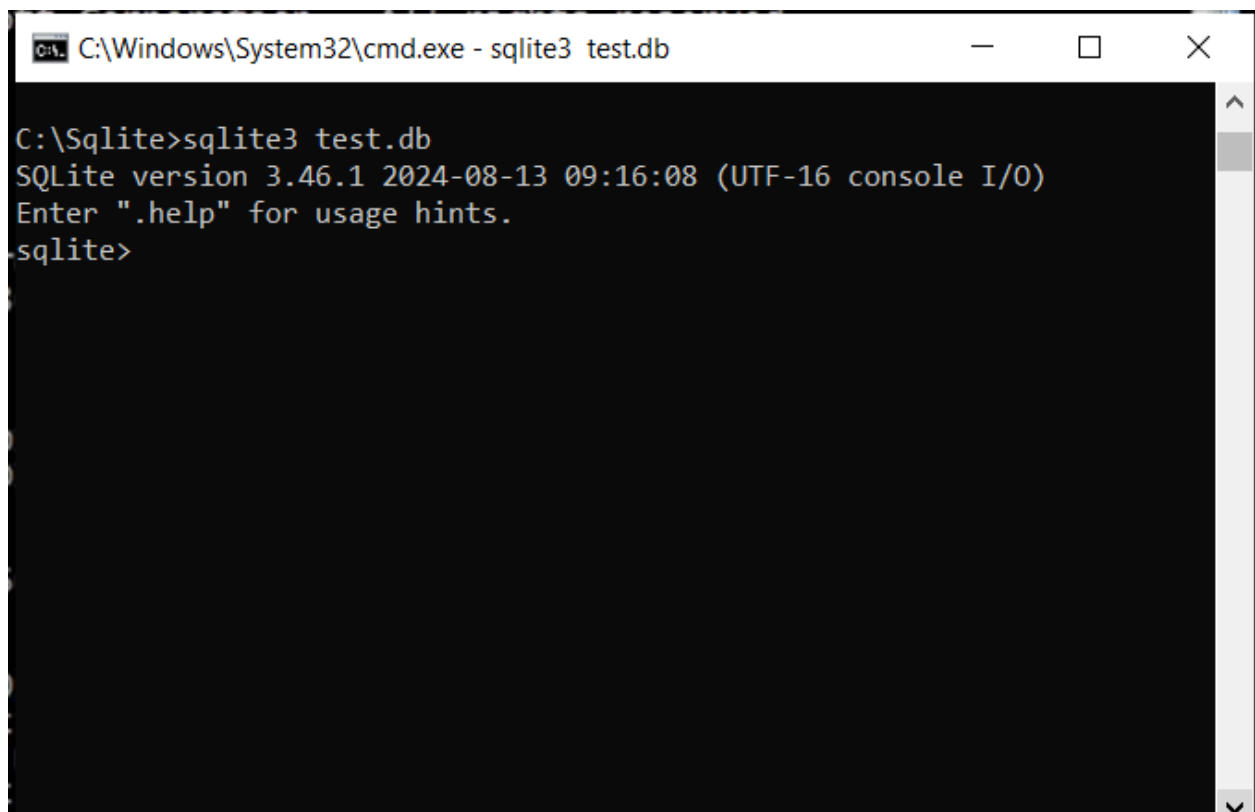
Always, database name should be unique within the RDBMS.

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.

A screenshot of a Windows command prompt window. The title bar reads "C:\Windows\System32\cmd.exe - sqlite3 test.db". The command prompt shows the following text: "C:\Sqlite>sqlite3 test.db", "SQLite version 3.46.1 2024-08-13 09:16:08 (UTF-16 console I/O)", "Enter ".help" for usage hints.", and "sqlite>". The prompt is on a black background with white text. A vertical scrollbar is visible on the right side of the command prompt window.

```
C:\Windows\System32\cmd.exe - sqlite3 test.db

C:\Sqlite>sqlite3 test.db
SQLite version 3.46.1 2024-08-13 09:16:08 (UTF-16 console I/O)
Enter ".help" for usage hints.
sqlite>
```

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
$
```

Create Table

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

Syntax

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
);
```

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*.

```
C:\Windows\System32\cmd.exe - sqlite3 test.db

C:\Sqlite>sqlite3 test.db
SQLite version 3.46.1 2024-08-13 09:16:08 (UTF-16 console I/O)
Enter ".help" for usage hints.
sqlite> create table company(id int primary key not null, name text not
null, age int not null, address char(50), salary real);
sqlite> _
```

```
C:\Windows\System32\cmd.exe - sqlite3 test.db

C:\Sqlite>sqlite3 test.db
SQLite version 3.46.1 2024-08-13 09:16:08 (UTF-16 console I/O)
Enter ".help" for usage hints.
sqlite> create table company(id int primary key not null, name text not
null, age int not null, address char(50), salary real);
sqlite> .tables
company
sqlite> create table department(id int primary key not null, dept char(5
0) not null, emp_id int not null);
sqlite> .tables
company      department
sqlite> _
```

Example

Following is an example which creates a COMPANY table with ID as the primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table.

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

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
);
```

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  
);
```

Conclusion: Thus we created a simple database and table.

Experiment No. 3

Title: Insert sample data into the table.

Aim: To Insert sample data into the table.

Theory:

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 sure the order of the values is in the same order as the columns in the table. The SQLite INSERT INTO syntax would be as follows –

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

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

```

C:\Windows\System32\cmd.exe - sqlite3 test.db
company
sqlite> create table department(id int primary key not null, dept char(50) not null, emp_id int not null);
sqlite> .tables
company      department
sqlite> insert into company(id,name,age,address,salary)values(1,sheetal,30,pune,2000)
...> ;
Parse error: no such column: sheetal
    into company(id,name,age,address,salary)values(1,sheetal,30,pune,2000) ;
              error here ---^
sqlite> insert into company(id,name,age,address,salary)values(1,'sheetal',30,'pune',2000);
sqlite> insert into company(id,name,age,address,salary)values(1,'mona',31,'mumbai',3000);
Runtime error: UNIQUE constraint failed: company.id (19)
sqlite> insert into company(id,name,age,address,salary)values(2,'mona',31,'mumbai',3000);
sqlite> insert into company(id,name,age,address,salary)values(3,'seema',32,'nasik',4000);
sqlite> insert into company(id,name,age,address,salary)values(4,'deepa',33,'nagpur',5000);
sqlite> insert into company(id,name,age,address,salary)values(5,'kiran',43,'amravati',6000);
sqlite>

C:\Windows\System32\cmd.exe - sqlite3 test.db
sqlite> insert into department(id,dept,emp_id)values(1,200,1);
sqlite> insert into department(id,dept,emp_id)values(2,200,2);
sqlite> insert into department(id,dept,emp_id)values(3,200,3);
sqlite> insert into department(id,dept,emp_id)values(4,200,4);
sqlite> insert into department(id,dept,emp_id)values(5,200,5);
sqlite>

```

Conclusion: Thus performed Inserting sample data into the table.

Experiment No. 4

Title: Perform basic SQL operations such as SELECT.

Aim: To Perform basic SQL operations such as SELECT.

Theory:

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;
```

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

```
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

```
C:\Windows\System32\cmd.exe - sqlite3 test.db

sqlite> select * from company;
1|sheetal|30|pune|2000.0
2|mona|31|mumbai|3000.0
3|seema|32|nasik|4000.0
4|deepa|33|nagpur|5000.0
5|kiran|43|amravati|6000.0
sqlite> select * from department;
1|200|1
2|200|2
3|200|3
4|200|4
5|200|5
sqlite>
```

Conclusion: Thus Performed basic SQL operations such as SELECT.

Experiment No. 5

Title: Perform basic SQL operations such as WHERE.

Aim: To Perform basic SQL operations such as WHERE.

Theory:

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 fetching only 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
WHERE [condition]
```



Example

You can specify a condition using Comparison or Logical Operators such as >, <, =, LIKE, NOT, etc. 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 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 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

```
C:\Windows\System32\cmd.exe - sqlite3 test.db
```

```
2|mona|31|mumbai|3000.0
3|seema|32|nasik|4000.0
4|deepa|33|nagpur|5000.0
5|kiran|43|amravati|6000.0
sqlite> select * from department;
```

1 200 1
2 200 2
3 200 3
4 200 4
5 200 5

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 2000;
```

1 sheetal 30 pune 2000.0
2 mona 31 mumbai 3000.0
3 seema 32 nasik 4000.0
4 deepa 33 nagpur 5000.0
5 kiran 43 amravati 6000.0

```
sqlite>
```

Conclusion: Thus Performed basic SQL operations such as WHERE

Experiment No. 6

Title: Perform basic SQL operations such as UPDATE.

Aim: To Perform basic SQL operations such as UPDATE.

Theory:

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

Syntax

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
7	James	24	Texas	20000.0


```
C:\Windows\System32\cmd.exe - sqlite3 test.db
4|200|4
5|200|5
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 2000;
1|sheetal|30|pune|2000.0
2|mona|31|mumbai|3000.0
3|seema|32|nasik|4000.0
4|deepa|33|nagpur|5000.0
5|kiran|43|amravati|6000.0
sqlite> UPDATE COMPANY SET ADDRESS = 'pune', SALARY = 20000.00;
sqlite> SELECT * FROM COMPANY
...> ;
1|sheetal|30|pune|20000.0
2|mona|31|pune|20000.0
3|seema|32|pune|20000.0
4|deepa|33|pune|20000.0
5|kiran|43|pune|20000.0
sqlite>
```

Conclusion: Thus Performed basic SQL operations such as UPDATE.

Experiment No. 7

Title: Perform basic SQL operations such as DELETE.

Aim: To Perform basic SQL operations such as DELETE.

Theory:

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.

```
C:\Windows\System32\cmd.exe - sqlite3 test.db
3|seema|32|nasik|4000.0
4|deepa|33|nagpur|5000.0
5|kiran|43|amravati|6000.0
sqlite> UPDATE COMPANY SET ADDRESS = 'pune', SALARY = 20000.00;
sqlite> SELECT * FROM COMPANY
...> ;
1|sheetal|30|pune|20000.0
2|mona|31|pune|20000.0
3|seema|32|pune|20000.0
4|deepa|33|pune|20000.0
5|kiran|43|pune|20000.0
sqlite> delete from company where age<33;
sqlite> select * from company
...> ;
4|deepa|33|pune|20000.0
5|kiran|43|pune|20000.0
sqlite>
```

Conclusion: Thus Performed basic SQL operations such as DELETE.

Experiment No. 8

Title: Querying SQLite Databases using ORDER BY.

Aim: To Query SQLite Databases using ORDER BY.

Theory:

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];
```

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

```

C:\Windows\System32\cmd.exe - sqlite3 test.db
sqlite> select * from company
...> ;
4|deepa|33|pune|20000.0
5|kiran|43|pune|20000.0
sqlite> select * from company order by name desc;
5|kiran|43|pune|20000.0
4|deepa|33|pune|20000.0
sqlite>

```

Conclusion: Thus performed Querying SQLite Databases using ORDER BY.

Experiment No. 9

Title: Querying SQLite Databases using GROUP BY.

Aim: To Query SQLite Databases using GROUP BY.

Theory:

SQLite **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups.

GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax

Following is the basic syntax of GROUP BY clause. GROUP BY clause must follow the conditions in the WHERE clause and must precede ORDER BY clause if one is used.

```

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 the 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

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

```
C:\Windows\System32\cmd.exe - sqlite3 test.db
qlite> select * from company
...> ;
|deepa|33|pune|20000.0
|kiran|43|pune|20000.0
qlite> select * from company order by name desc;
|kiran|43|pune|20000.0
|deepa|33|pune|20000.0
qlite> insert into company(id,name,age,address,salary)values(1,'mona',31,'mumbai',3000);
qlite> insert into company(id,name,age,address,salary)values(2,'mona',31,'mumbai',3000);
qlite> insert into company(id,name,age,address,salary)values(3,'seema',32,'nasik',4000);
qlite> select * from company;
|deepa|33|pune|20000.0
|kiran|43|pune|20000.0
|mona|31|mumbai|3000.0
|mona|31|mumbai|3000.0
|seema|32|nasik|4000.0
qlite>
```

```
C:\Windows\System32\cmd.exe - sqlite3 test.db
4|deepa|33|pune|20000.0
sqlite> insert into company(id,name,age,address,salary)values(1,'mona',31,'mumbai',3000);
sqlite> insert into company(id,name,age,address,salary)values(2,'mona',31,'mumbai',3000);
sqlite> insert into company(id,name,age,address,salary)values(3,'seema',32,'nasik',4000);
sqlite> select * from company;
4|deepa|33|pune|20000.0
5|kiran|43|pune|20000.0
1|mona|31|mumbai|3000.0
2|mona|31|mumbai|3000.0
3|seema|32|nasik|4000.0
sqlite> SELECT NAME, SUM(SALARY)
...> FROM COMPANY GROUP BY NAME ORDER BY NAME DESC;
seema|4000.0
mona|6000.0
kiran|20000.0
deepa|20000.0
sqlite>
```

Conclusion: Thus performed Querying SQLite Databases using GROUP BY.

Experiment No. 10

Title: Querying SQLite Databases using JOINS.

Aim: To Perform basic SQL operations such as JOINS.

Theory:

The INNER JOIN

INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with 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.

Following is the syntax of INNER JOIN –

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

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

```
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

```
C:\Windows\System32\cmd.exe - sqlite3 test.db
|mona|31|mumbai|3000.0
|mona|31|mumbai|3000.0
|seema|32|nasik|4000.0
qlite> SELECT NAME, SUM(SALARY)
...> FROM COMPANY GROUP BY NAME ORDER BY NAME DESC;
seema|4000.0
mona|6000.0
kiran|20000.0
deepa|20000.0
qlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT
...> ON COMPANY.ID = DEPARTMENT.EMP_ID;
|mona|200
|mona|200
|seema|200
|deepa|200
|kiran|200
qlite>
```

Conclusion: Thus we performed SQL operations such as JOINS.