Database access using SQLite

Dr Venkat Raju

Data

Quantities 1,2,3,…100,…1000…

Characters A, B, C…Z, a, b, c,…z

Symbols ! @ # $ % & * ()…

Quantities , Characters, Symbols are stored in digital format.

Data → plural Datum → singular

Data is Every Where

Machines, Robots, Sensors, Our self are products of data.

All roads lead to DATA.

Numerous kinds of data

Text data (.doc, .txt, .pdf…)

Excel data (.csv, .tsv)

HTML data

XML data

JSON data

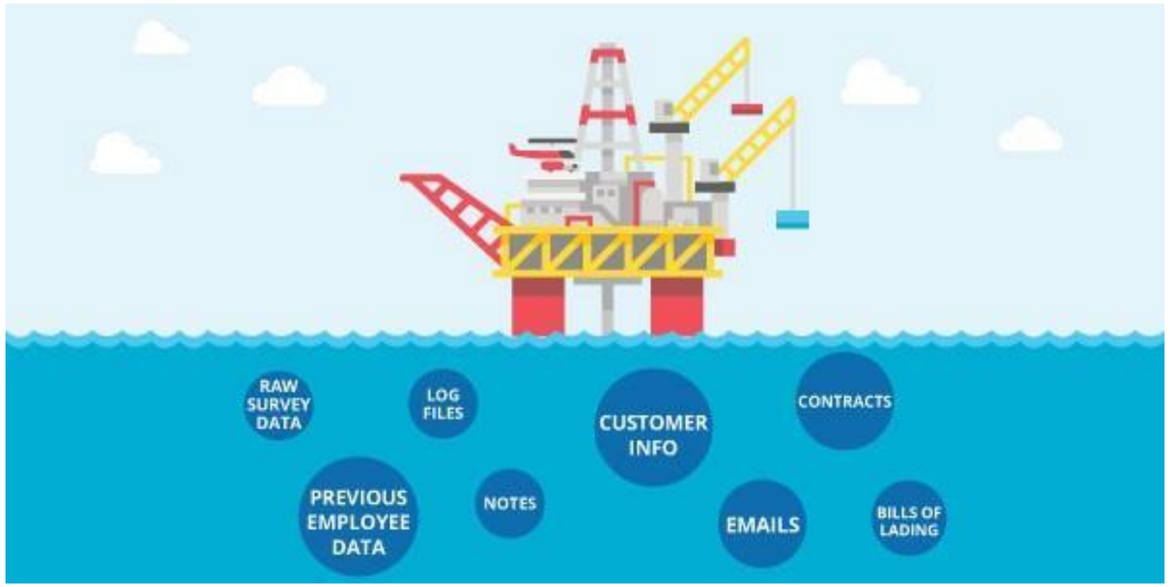
Relational Database (Oracle, MySql, Sqlite…)

Log files / Transactional data

Sensors/Web servers

Social Media data (FB, Twitter, WhatsApp, YouTube…)

Image / Audio / Video / Signal….



Database

- A **Database** is a shared collection of logically related data and description of these data, designed to meet the information needs of an organization
- A **Database Management System** is a software system that enables users to define, create, maintain, and control access to the database.

Various Database systems

- There are many different database systems which are used for a wide variety of purposes including: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, and SQLite.
- SQL is a programming language used by nearly all relational databases to

  query,

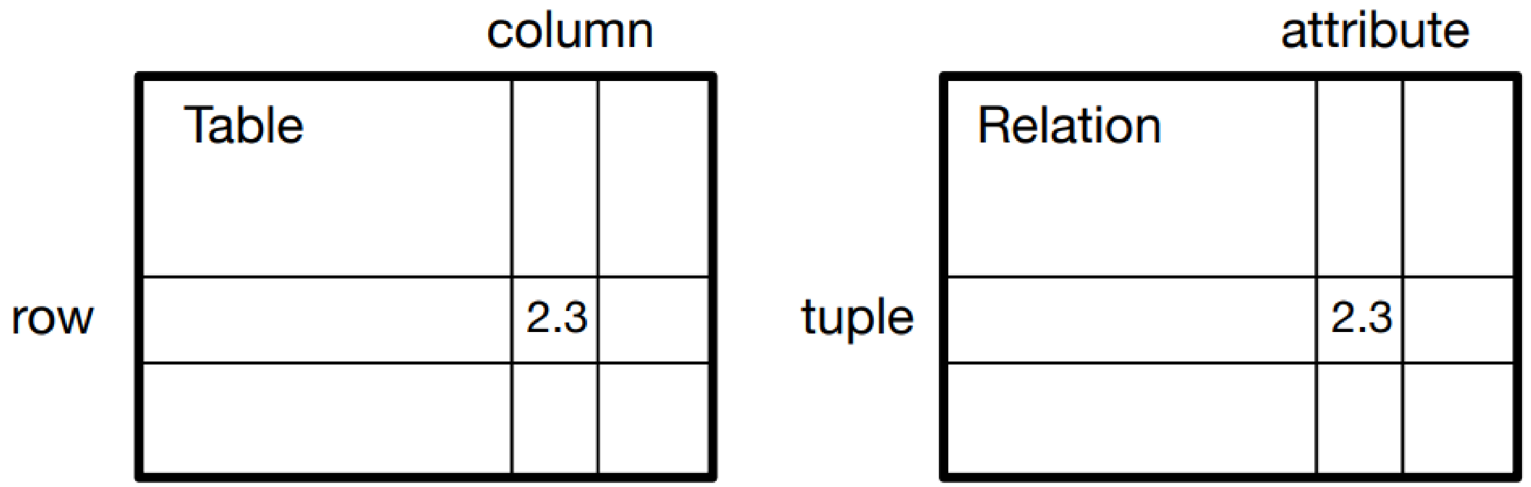
  manipulate, and

  define data, and

  to provide access control.

Relational Databases

- Relational Databases store data in relations i.e., tables. Each relation must have a name.
- The primary data structures in a database : **tables, rows, and columns.**
- In technical descriptions of relational databases, the concepts of **table**, **row**, and **column** are formally referred to as **relation**, **tuple**, and **attribute**, respectively.



Working with Databases

- A database is an abstraction over an operating system's file system that makes it easier for developers to build applications that create, read, update and delete persistent data.
- At a high-level web applications store data and present it to users in a useful way.

  Example

  - Google stores data about roads and provides directions to get from one location to another by driving through the Maps application.
  - Driving directions are possible because the data is stored in a structured format.
- Databases make structured storage reliable and fast.

Working with Databases

- The database storage abstraction most used in Python web development is sets of relational tables.
    - Relational databases store data in a series of tables. Interconnections between the tables are specified as *foreign keys*.
    - A foreign key is a unique reference from one row in a relational table to another row in a table, which can be the same table but is most commonly a different table.
- Databases storage implementations vary in complexity. **SQLite**, a database included with Python, creates a single file for all data per database.
    - Other databases such as PostgreSQL, MySQL, Oracle and Microsoft SQL Server have more complicated persistence schemes while offering additional advanced features that are useful for web application data storage.

SQLite database

- SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world.
    - SQLite does not require a separate server process or system to operate (serverless).
    - SQLite comes with zero-configuration, which means no setup or administration needed.
    - A complete SQLite database is stored in a single cross-platform disk file.
    - SQLite is very small and light weight, less than 400KiB fully configured or less than 250KiB with optional features omitted.
    - SQLite is self-contained, which means no external dependencies.
    - SQLite transactions are fully ACID-compliant, allowing safe access from multiple processes or threads.
    - Supports in all O.S, written in ANSI-C and provides simple and easy-to-use API.

ACID properties

- **Atomicity**
  - **A**tomic: 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**:
  - when the transaction is committed or rolled back, it is important that the transaction must keep the database consistent.
- **Isolation**
  - In case if two transactions executing concurrently only respective transaction results will appear to the client, not other client's transaction results.
- **Durability**
  - Durability means once the transaction committed successfully then it is guaranteed that all the changes committed permanently to the database

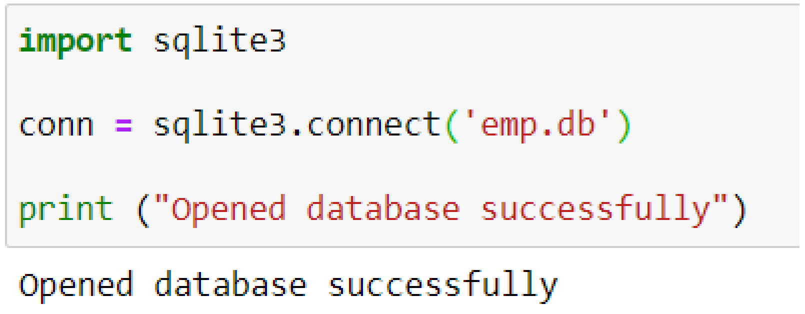
How to get start with databases

- Databases require more defined structure than Python lists or dictionaries.
- When we create a database *table*, we must tell the database in advance the names of each of the *columns* in the table and the type of data which we are planning to store in each *column*.
- When the database software knows the type of data in each column, it can choose the most efficient way to store and look up the data based on the type of data.

How to work with SQLite

- **SQLite database is in-built in Python** using **sqlite3 module**
- When we connect to an SQLite database file that does not exist, SQLite automatically creates the new database for us.
- To create a database, first, we must create a Connection object that represents the database using the connect() function of the sqlite3 module.

```
import sqlite3
conn = sqlite3.connect('emp.db')
print ("Opened database successfully")
```



```
import sqlite3

conn = sqlite3.connect('emp.db')

print ("Opened database successfully")
```

```
Opened database successfully
```

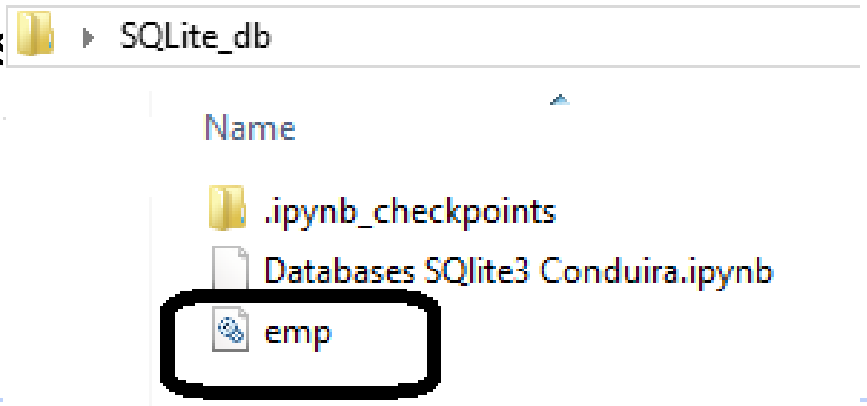
Steps to work with SQLite

Note: We must create a folder first,

before we execute the program. or

```
import sqlite3
conn = sqlite3.connect('emp.db')
print ("Opened database successfully"
```

- we can place the database file a folder of our choice.
- The connect() function opens a connection to an SQLite database. It returns a Connection object that represents the database. By using the Connection object, you can perform variou



SQLite_db

Name

.ipynb_checkpoints
Databases SQlite3 Conduira.ipynb
emp

Steps to work with SQLite

Note:

- If we skip the folder path C:\Users\PERSONAL\Desktop\SQLite_db, the program will create the database file in the current working directory (CWD).
- If we pass the file name as **:memory:** to the connect() function of the sqlite3 module, it will create a new database that resides in the memory (RAM) instead of a database file on disk.

```
import sqlite3

conn = sqlite3.connect(':memory:')

print ("Opened database successfully")
```

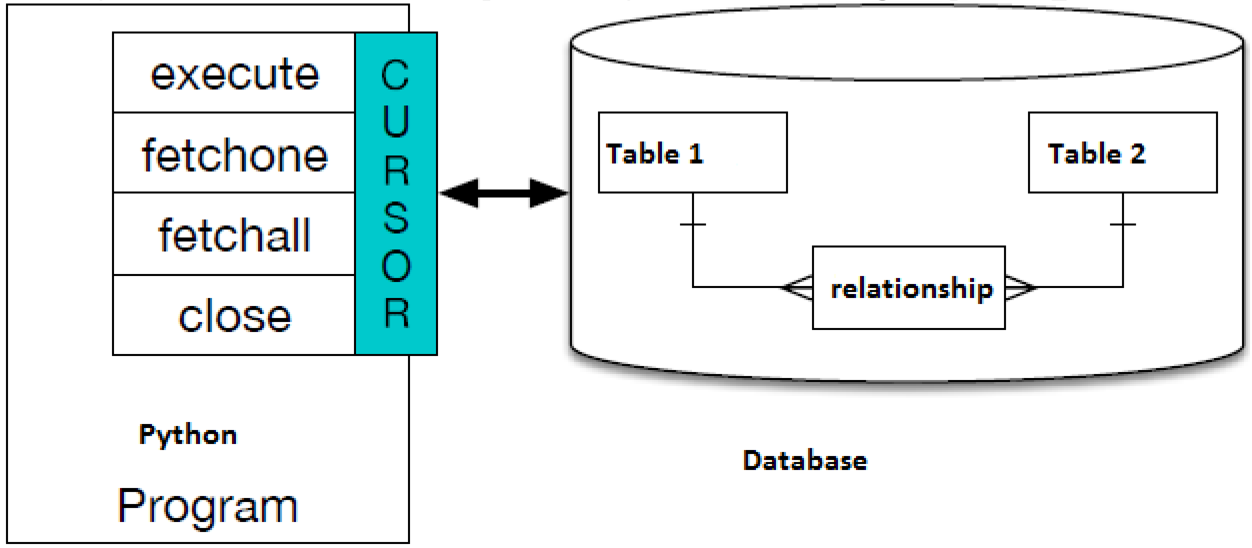
Creating Tables in SQLite

- To create a new table in an SQLite database from a Python program, you use the following steps:
  - First, create a **Connection** object using the **connect()** method of the sqlite3 module.
  - Second, create a **Cursor object** by calling the **cursor()** method of the Connection object.
  - Third, pass the **CREATE TABLE** statement to the **execute**() method of the Cursor object and execute this method.

Database cursors

- A *cursor* is like a file handle that we can use to perform operations on the data stored in the database.

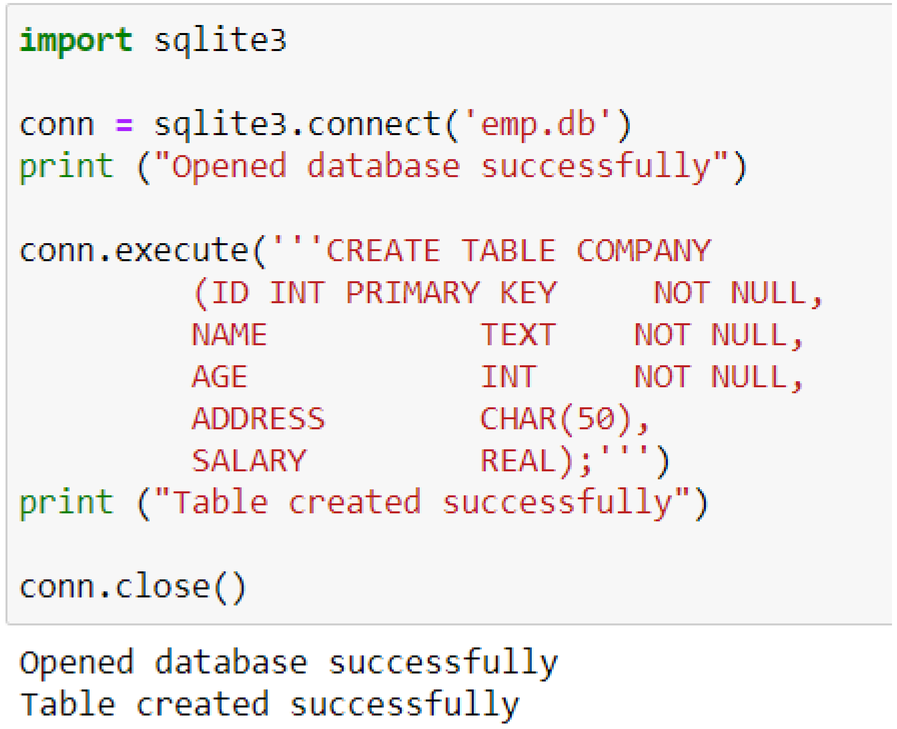
- Calling cursor() is very similar conceptually to calling open() when dealing with text files.



- Once we have the cursor, we can begin to execute commands on the contents of the database using the execute() method.

Creating Tables in SQLite

```
import sqlite3

conn = sqlite3.connect('emp.db')
print ("Opened database successfully")

conn.execute('''CREATE TABLE COMPANY
       (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(50),
       SALARY         REAL);''')
print ("Table created successfully")

conn.close()
```



```
import sqlite3

conn = sqlite3.connect('emp.db')
print ("Opened database successfully")

conn.execute('''CREATE TABLE COMPANY
         (ID INT PRIMARY KEY     NOT NULL,
         NAME           TEXT    NOT NULL,
         AGE            INT     NOT NULL,
         ADDRESS        CHAR(50),
         SALARY         REAL);''')
print ("Table created successfully")

conn.close()
```

```
Opened database successfully
Table created successfully
```

CURD operations using SQLite

CURD operations using SQLite

- CURD stands for CREATE, UPDATE, RETRIEVE, DELETE operations.
    - Create – Insert operation
    - Update- Update operation
    - Retrieve – Select operation
    - Delete – Delete operation

Insert data into SQLite

```
import sqlite3

conn = sqlite3.connect('emp.db')
print ("Opened database successfully")

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)VALUES (1, 'Paul', 32, 'California', 20000.00 )");

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

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

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

conn.commit()
print ("Records created successfully")
conn.close()
```

```
Opened database successfully
Records created successfully
```

Select data from SQLite

```
import sqlite3

conn = sqlite3.connect('emp.db')
print ("Opened database successfully")

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print ("ID = ", row[0])
   print ("NAME = ", row[1])
   print ("ADDRESS = ", row[2])
   print ("SALARY = ", row[3], "\n")

print ("Operation done successfully")
conn.close()
```

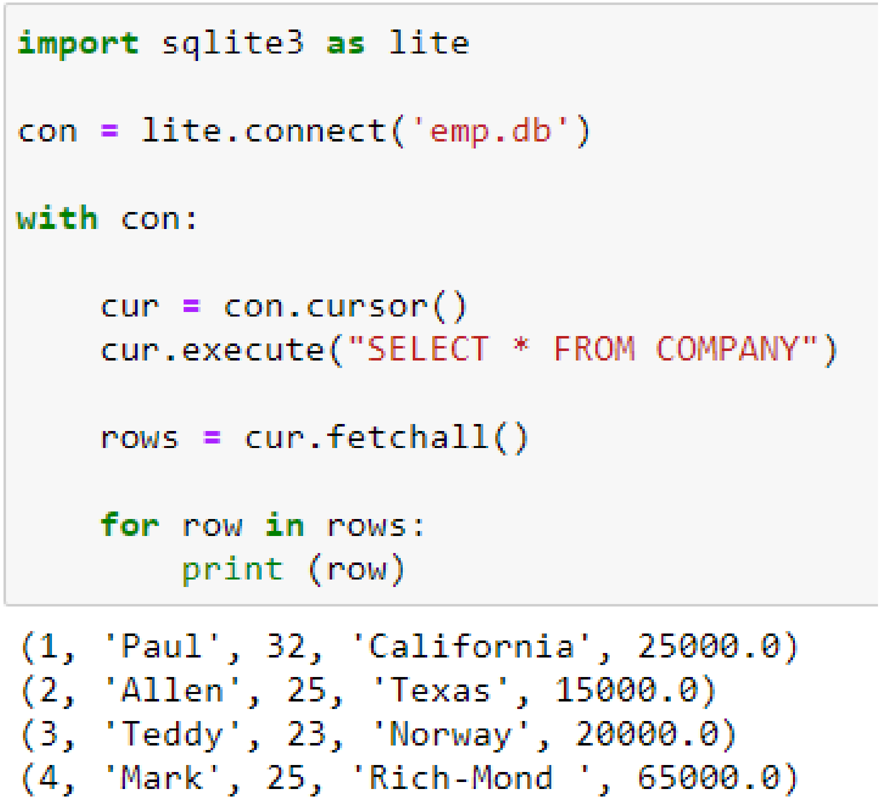
```
Opened database successfully
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  25000.0

ID =  2
NAME =  Allen
ADDRESS =  Texas
SALARY =  15000.0
..
..

ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0

Operation done successfully
```

Cursor methods

- cursor.fetchall() fetches all the rows of a query result.
  - It returns all the rows as a list of tuples.
  - An empty list is returned if there is no record to fetch.
- cursor.fetchmany(size) returns the number of rows specified by size argument.
  - When called repeatedly this method fetches the next set of rows of a query result and returns a list of tuples.
  - If no more rows are available, it returns an empty list.
- cursor.fetchone() method returns a single record or None if no more rows are available.

Select data from SQLite

```
import sqlite3 as lite

con = lite.connect('emp.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM COMPANY")

    rows = cur.fetchall()

    for row in rows:
        print (row)
```

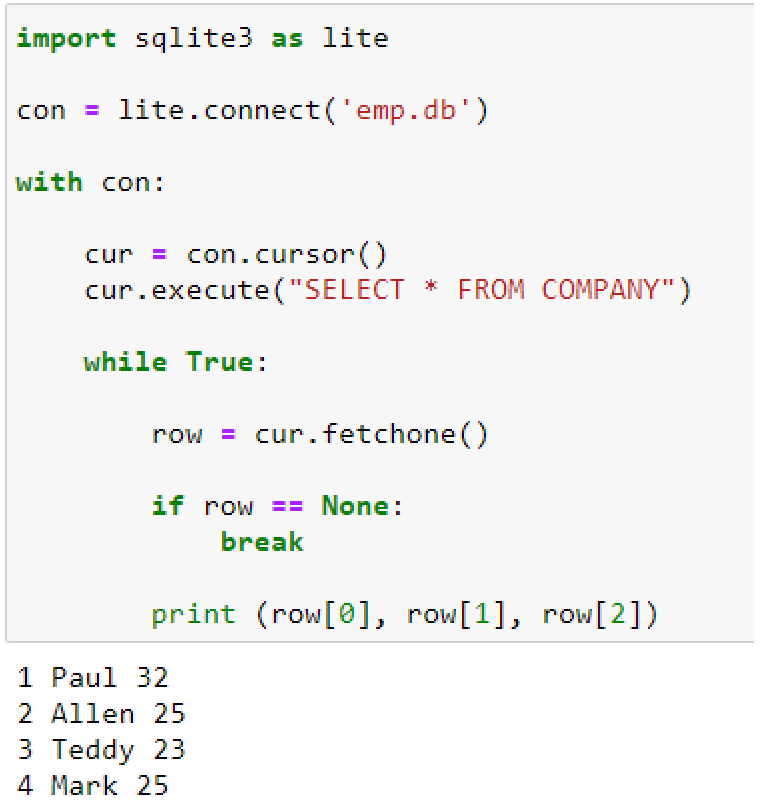


```
import sqlite3 as lite

con = lite.connect('emp.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM COMPANY")

    rows = cur.fetchall()

    for row in rows:
        print (row)
```

```
(1, 'Paul', 32, 'California', 25000.0)
(2, 'Allen', 25, 'Texas', 15000.0)
(3, 'Teddy', 23, 'Norway', 20000.0)
(4, 'Mark', 25, 'Rich-Mond ', 65000.0)
```

Retrieving one record at a time

```
import sqlite3 as lite

con = lite.connect('emp.db')

with con:
    cur = con.cursor()
    cur.execute("SELECT * FROM COMPANY")

    while True:
        row = cur.fetchone()

        if row == None:
            break

        print (row[0], row[1], row[2])
```

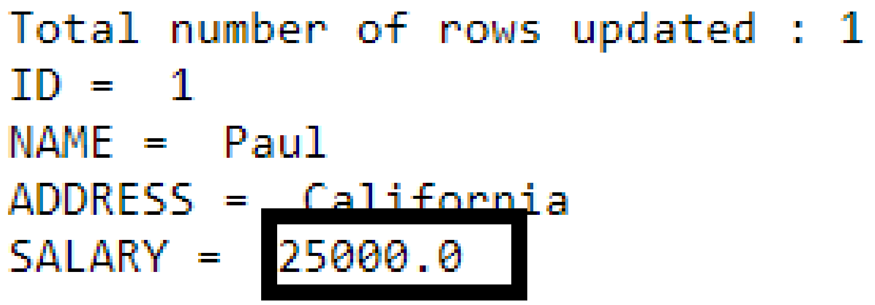


```
import sqlite3 as lite

con = lite.connect('emp.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM COMPANY")

    while True:

        row = cur.fetchone()

        if row == None:
            break

        print (row[0], row[1], row[2])
```

```
1 Paul 32
2 Allen 25
3 Teddy 23
4 Mark 25
```

Update SQLite data

```
import sqlite3

conn = sqlite3.connect('emp.db')

conn.execute("UPDATE COMPANY set SALARY = 25000.00 where ID = 1")
conn.commit()
print ("Total number of rows updated :", conn.total_changes)

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print ("ID = ", row[0])
   print ("NAME = ", row[1])
   print ("ADDRESS = ", row[2])
   print ("SALARY = ", row[3], "\n")

conn.close()
```



```
Total number of rows updated : 1
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  25000.0
```

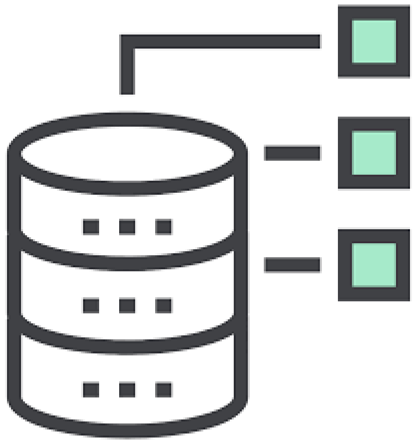
Deleting data from SQLite

```
import sqlite3

conn = sqlite3.connect('emp.db')
print ("Opened database successfully")

conn.execute("DELETE from COMPANY where ID = 2;")
conn.commit()
print ("Total number of rows deleted :", conn.total_changes)

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print ("ID = ", row[0])
   print ("NAME = ", row[1])
   print ("ADDRESS = ", row[2])
   print ("SALARY = ", row[3], "\n")

print ("Operation done successfully")
conn.close()
```

```
Opened database successfully
Total number of rows deleted : 1
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  25000.0

ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0

ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0

Operation done successfully
```

Database access using SQLite

Part 2



Dr Venkat Raju

Parameterized queries

- A parameterized query is a **query in which placeholders used for parameters and the parameter values supplied at execution time**.
- That means parameterized query gets compiled only once.
- **There are the main 4 reasons to use**
  - Improves Speed: If you want to execute SQL statement/query many times, it usually reduces execution time
  - Compile Once: The main advantage of using a parameterized query is that parameterized query compiled only once
  - Same Operation with Different Data: if you want to execute the same query multiple times with different data.
  - Preventing SQL injection attacks.

Parameterized queries

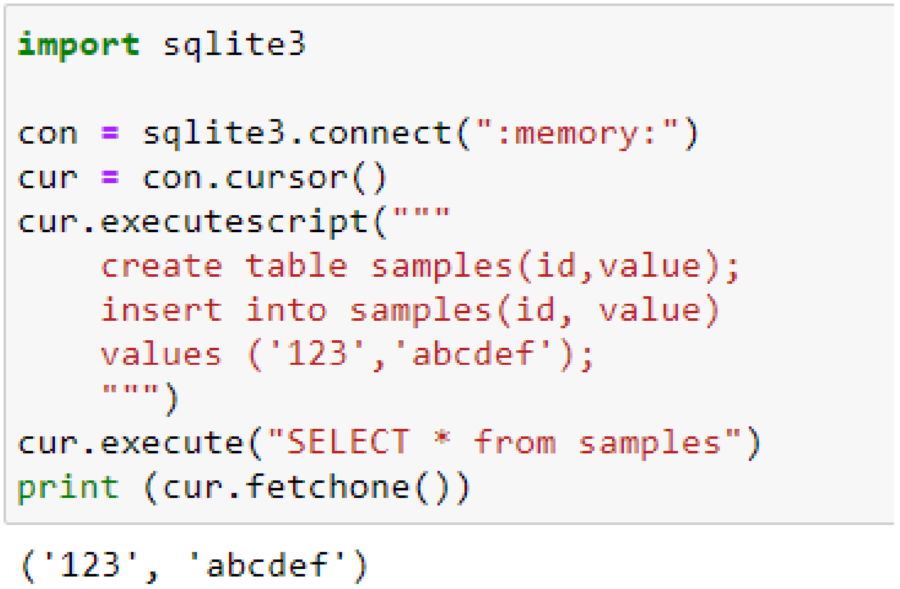
```
import sqlite3
conn = sqlite3.connect('LanguageDB')
cur = conn.cursor()
cur.execute('DROP TABLE IF EXISTS languages')
cur.execute('CREATE TABLE languages (subject TEXT, marks INTEGER)')

cur.execute('INSERT INTO languages (subject, marks) VALUES (?, ?)',  ('C', 100))
cur.execute('INSERT INTO languages (subject, marks) VALUES (?, ?)',  ('Java', 200))
cur.execute('INSERT INTO languages (subject, marks) VALUES (?, ?)',  ('Python', 300))
conn.commit()

print('Languages:')
cur.execute('SELECT subject, marks FROM languages')
for row in cur:
    print(row)
cur.close()
```

```
Languages:
('C', 100)
('Java', 200)
('Python', 300)
```

executescript() - executing multiple SQL statements

- This is a nonstandard convenience method for executing multiple SQL statements at once. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter.

```
import sqlite3
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table samples(id,value);
    insert into samples(id, value)  values ('123','abcdef');
    """)
cur.execute("SELECT * from samples")
print (cur.fetchone())
```



```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table samples(id,value);
    insert into samples(id, value)
    values ('123','abcdef');
    """)
cur.execute("SELECT * from samples")
print (cur.fetchone())
```

```
('123', 'abcdef')
```

Consider database as emp.db

emp10.db

```
(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)
```

select with where clause

```
import sqlite3 as lite

con = lite.connect('emp10.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM COMPANY where ID=1")

    rows = cur.fetchall()

    for row in rows:
        print (row)
```

```
(1, 'Paul', 32, 'California', 20000.0)
```

select with where clause

```
import sqlite3 as lite

con = lite.connect('emp0.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM COMPANY where NAME='Paul'")

    rows = cur.fetchall()

    for row in rows:
        print (row)
```

```
(1, 'Paul', 32, 'California', 25000.0)
```

select with where clause

```
import sqlite3 as lite

con = lite.connect('emp10.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM COMPANY where SALARY >=30000")

    rows = cur.fetchall()

    for row in rows:
        print (row)
```

```
(4, 'Mark', 25, 'Rich-Mond ', 65000.0)
```

select with *order by* clause

```
import sqlite3 as lite

con = lite.connect('emp10.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM COMPANY ORDER BY  SALARY")

    rows = cur.fetchall()

    for row in rows:
        print (row)
```

```
(2, 'Allen', 25, 'Texas', 15000.0)
(1, 'Paul', 32, 'California', 20000.0)
(3, 'Teddy', 23, 'Norway', 20000.0)
(4, 'Mark', 25, 'Rich-Mond ', 65000.0)
```

select with where clause (like operator)

```
import sqlite3 as lite

con = lite.connect('emp10.db')

with con:


    cur = con.cursor()
    cur.execute("SELECT * FROM COMPANY WHERE NAME LIKE 'a%'")

    rows = cur.fetchall()

    for row in rows:
        print (row)
```

```
(2, 'Allen', 25, 'Texas', 15000.0)
```

select with where clause (max())

```
import sqlite3 as lite

con = lite.connect('emp10.db')

with con:


    cur = con.cursor()
    cur.execute("SELECT MAX(salary) FROM COMPANY")

    rows = cur.fetchall()

    for row in rows:
        print (row)
```

```
(65000.0,)
```

Working with JOINS

Working with Joins using SQLite

- SQLite JOINS are used to retrieve data from multiple tables.
- An SQLite JOIN is performed whenever two or more tables are joined in a SQL statement.
- There are different types of SQLite joins:
  - INNER JOIN (or sometimes called simple join)
  - LEFT OUTER JOIN (or sometimes called LEFT JOIN)
  - CROSS JOIN

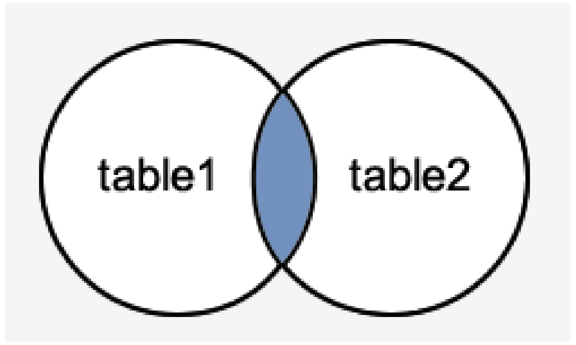
- **Note : SQLite doesn’t directly support the RIGHT JOIN and FULL OUTER JOIN.**

Inner Join

- SQLite INNER JOINS return all rows from multiple tables where the join condition is met.

The syntax for the INNER JOIN in SQLite is:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```



In this visual diagram, the SQLite INNER JOIN returns the shaded area:

```
SELECT employees.employee_id,

employees.last_name, positions.title

FROM employees

INNER JOIN positions

ON employees.position_id = positions.position_id;
```

return all rows from the *employees* and *positions* tables where there is a matching *position_id* value in both the *employees* and *positions* tables.

Inner Join

employees

employee_id	last_name	first_name	position_id
10000	Smith	John	1
10001	Anderson	Dave	2
10002	Doe	John	3
10003	Hunt	Dylen	

positions

position_id	title
1	Manager
2	Project Planner
3	Programmer
4	Data Analyst

Result table

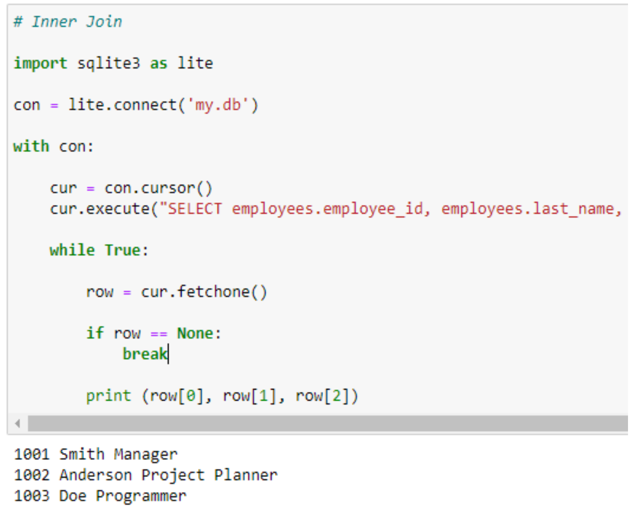
employee_id	last_name	title
10000	Smith	Manager
10001	Anderson	Project Planner
10002	Doe	Programmer

- The row for *employee_id* 10003 from the *employees* table would be omitted, since the corresponding *position_id* does not exist in the *positions* table.

- The row for the *position_id* of 4 from the *positions* table would be omitted, since that *position_id* does not exist in the *employees* table.

Implementation of inner join

```
# Inner Join
import sqlite3 as lite
con = lite.connect('my.db')

with con:
    cur = con.cursor()
    cur.execute("SELECT employees.employee_id, employees.last_name,
    positions.title FROM  employees INNER JOIN positions ON
    employees.position_id = positions.position_id;")

    while True:
        row = cur.fetchone()

        if row == None:
            break

        print (row[0], row[1], row[2])
```



```
# Inner Join

import sqlite3 as lite

con = lite.connect('my.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT employees.employee_id, employees.last_name,

    while True:

        row = cur.fetchone()

        if row == None:
            break

        print (row[0], row[1], row[2])
```

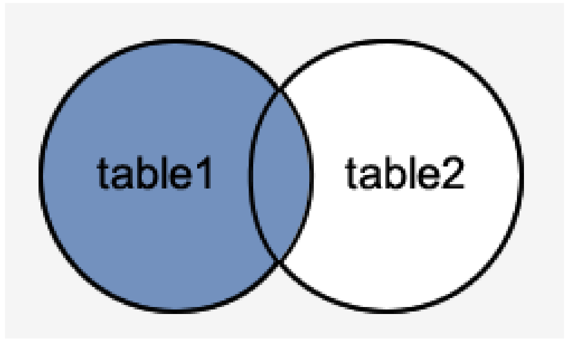
```
1001 Smith Manager
1002 Anderson Project Planner
1003 Doe Programmer
```

LEFT OUTER JOIN

- Outer join returns all rows from the LEFT-hand table specified in the ON condition and only those rows from the other table where the joined fields are equal (join condition is met).

The syntax for the SQLite LEFT OUTER JOIN is:

```
SELECT columns
FROM table1
LEFT [OUTER] JOIN table2
ON table1.column = table2.column;
```



```
SELECT employees.employee_id, employees.last_name,
positions.title
FROM employees
LEFT OUTER JOIN positions
ON employees.position_id = positions.position_id;
```

The SQLite LEFT OUTER JOIN would return the all records from *table1* and only those records from *table2* that intersect with *table1*.

LEFT OUTER JOIN

employees

employee_id	last_name	first_name	position_id
10000	Smith	John	1
10001	Anderson	Dave	2
10002	Doe	John	3
10003	Hunt	Dylen	

positions

position_id	title
1	Manager
2	Project Planner
3	Programmer
4	Data Analyst

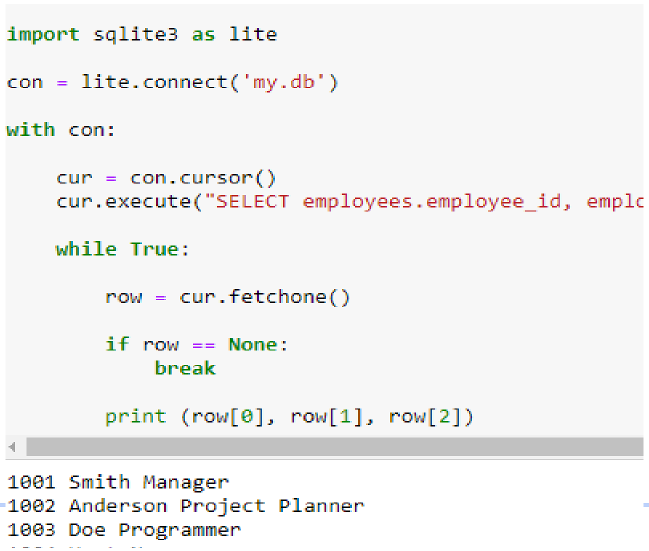
Result table

employee_id	last_name	title
10000	Smith	Manager
10001	Anderson	Project Planner
10002	Doe	Programmer
10003	Hunt	<null>

- The row for *employee_id* 10003 would be included because a LEFT OUTER JOIN was used.
- However, you will notice that the *title* field for that record contains a <null> value since there is no corresponding row in the *positions* table.

Implementation of LEFT OUTER JOIN

```
import sqlite3 as lite
con = lite.connect('my.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT employees.employee_id, employees.last_name, positions.title

    FROM employees LEFT OUTER JOIN positions ON employees.position_id = positions.position_id;")

    while True:
        row = cur.fetchone()

        if row == None:
            break

        print (row[0], row[1], row[2])
```



```
import sqlite3 as lite

con = lite.connect('my.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT employees.employee_id, emplo

    while True:

        row = cur.fetchone()

        if row == None:
            break

        print (row[0], row[1], row[2])
```

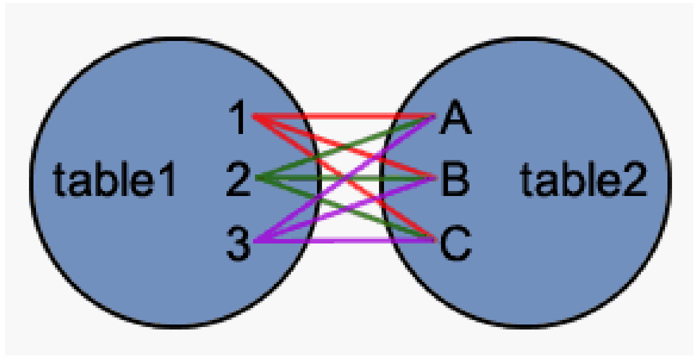
```
1001 Smith Manager
1002 Anderson Project Planner
1003 Doe Programmer
```

CROSS JOIN

- This type of join returns a combined result set with every row from the first table matched with every row from the second table. This is also called a Cartesian Product.

The syntax for the SQLite CROSS JOIN is:

```
SELECT columns
FROM table1
CROSS JOIN table2;
```



CROSS JOIN

position_id	title
1	Manager
2	Project Planner
3	Programmer
4	Data Analyst

department_id	department_name
30	HR
999	Sales

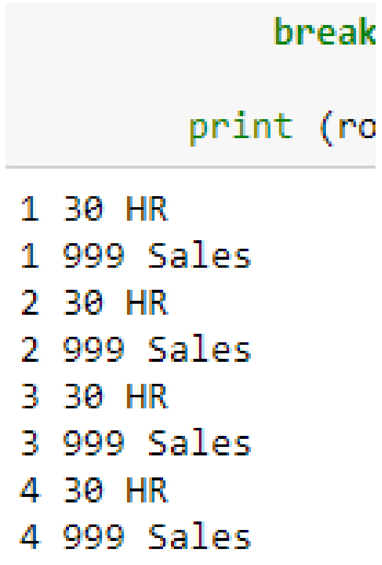
position_id	title	department_id	department_name
1	Manager	30	HR
1	Manager	999	Sales
2	Project Manager	30	HR
2	Project Manager	999	Sales
3	Programmer	30	HR
3	Programmer	999	Sales
4	Data Analyst	30	HR
4	Data Analyst	999	Sales

Since the *positions* table has 4 rows and the *departments* has 2 rows, the cross join will return 8 rows (because 4x2=8).

Each row from the *positions* table is matched with each row from the *departments* table.

Implementation of CROSS JOIN

```
import sqlite3 as lite
con = lite.connect('my.db')

with con:
    cur = con.cursor()
    cur.execute("SELECT position_id, department_id,

    department_name FROM positions CROSS JOIN departments;")

    while True:

        row = cur.fetchone()

        if row == None:
            break

        print (row[0], row[1], row[2])
```



```
            break

        print (ro
```

```
1 30 HR
1 999 Sales
2 30 HR
2 999 Sales
3 30 HR
3 999 Sales
4 30 HR
4 999 Sales
```

Transactions using SQLite

Transactions

- 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.
- The following commands are used to control transactions.
  - COMMIT − to save the changes.
  - ROLLBACK − to roll back the changes.
  - SAVEPOINT − creates points within the groups of transactions in which to ROLLBACK.

Creating a Transaction and rollback()

```
import sqlite3 as lite
import sys

con = lite.connect('test.db')
#con = lite.connect('test.db', isolation_level=None)

cur = con.cursor()

cur.execute("CREATE TABLE temp(id INTEGER PRIMARY KEY, name TEXT)")
cur.execute("INSERT INTO temp(name) VALUES ('Tom')")
cur.execute("INSERT INTO temp(name) VALUES ('Rebecca')")
cur.execute("INSERT INTO temp(name) VALUES ('Jim')")
cur.execute("INSERT INTO temp(name) VALUES ('Robert')")

#con.commit()

con.rollback()
con.close()
```

```
import sqlite3 as lite

con = lite.connect('test.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM temp")

    while True:

        row = cur.fetchone()

        if row == None:
            break

        print (row[0], row[1])
```