# SQLite – The Lightweight and Quick Response Database-Pandas

SQL is a language every analyst and data scientist should know. There’s no escaping from this. You will be peppered with SQL questions in your analytics or data science interview rounds, especially if you’re a fresher in this field.

If you’ve been putting off learning SQL recently, it’s time to get into action and start getting your hands dirty. You would have to learn about databases to work with data so why not start your SQL journey today?



And given the proliferation of data all over the world, every business is looking for professionals who are proficient in SQL. So once you add SQL skill to your resume, you will be a hotshot commodity out in the market. Great, but where to begin?

There are many different database systems out there, but the simplest and easiest to work with is **SQLite**. It is fast, compact, and stores data in an easy to share file format. It is used inside countless mobile phones, computers, and various other applications used by people every day. And the most amazing part, it comes bundled with Python! Heck, there is a reason why giants like Facebook, Google, Dropbox, and others use SQLite!

We will explore how to work with databases in Python using SQLite and look into the most commonly used SQL commands. So let’s start by asking the very basic question – what on earth is a database?

# Main Points

* What is a Database?
* What is SQL?
* Why Should you use SQLite?
* Connecting to an SQLite Database
* Creating tables using SQL
* Inserting values in a table using SQL
* Fetching records from a table using SQL
* Loading a Pandas DataFrame into SQLite Database
* Reading an SQLite Database into a Pandas DataFrame
* Querying SQLite Database
  + Where clause
  + Group By clause
  + Order By clause
  + Having clause
  + Join clause
  + Update statement

# What is a Database?

*A****database****is an organized collection of interrelated data stored in an electronic format.*

It is structured so as to provide easy access and manipulation of the stored data. Organizations use it to store any information that may be necessary for the decision-making process. The major advantages of databases over normal file storage systems are that it reduces data redundancy to a large extent, facilitates sharing of data among various users, and ensures the security of data which may be of immense importance to an organization.

While there are various types of databases and their choice of usage varies from organization to organization, the most basic and widely used is the**Relational Database model.** It organizes the data into tables where each row holds a record and is called a **tuple**. And each column represents an **attribute** for which each record usually holds a value.

A Relational database breaks down different aspects of a problem into different tables so that storing them and manipulating them becomes an easy task. For example, an e-commerce website maintaining a separate table for products and customers will find it more useful for doing analytics than saving all of the information in the same table.

**Database Management System (DBMS)** is a software that facilitates users and different applications to store, retrieve, and manipulate data in a database. **Relational Database Management System or RDBMS** is a DBMS for relational databases. There are many RDBMS like MYSQL, Postgres, SQL Server, etc. which use SQL for accessing the database.

# What is SQL?

But wait – we’ve been hearing the word ‘SQL’ since the beginner. What in the world is SQL?

**SQL**stands for **Structured Query Language**. It is a querying language designed for accessing and manipulating information from RDBMS.

SQL lets us write queries or sets of instructions to either create a new table, manipulate data or query on the stored data. Being a data scientist, it becomes imperative for you to know the basics of SQL to work your way around databases because you can only perform analysis if you can retrieve data from your organization’s database!

# Why Should you use SQLite?

[**SQLite**](https://www.sqlite.org/index.html) is a relational database management system based on SQL. It is designed for embedded devices that require fast and reliable data. It is serverless, lightweight, and requires zero-configuration. Also, it reads and writes directly to a disk file that can be easily copied and is platform-independent.

SQLite stores data in variable-length records which requires less memory and makes it run faster. It is designed for improved performance, reduced cost, and optimized for concurrency.

The [**sqlite3**](https://docs.python.org/2/library/sqlite3.html) module facilitates the use of SQLite databases with **Python**. In this article, I will show you how to work with an SQLite database in Python. You don’t need to download SQLite as it is shipped by default along with Python version 2.5 onwards!

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/05/SQLite.png)

# Creating tables using SQL

Now that we have created a database, it is time to create a table to store values.

Let’s create a table that stores values for a customer of an e-commerce website. It stores values like customer name, the id of the product bought, name, gender, age, and the city the customer is from.

A table in SQL is created using the **CREATE TABLE** command. Here I am going to create a table called *Customer*with the following attributes:

* User\_ID – Id to identify individual customers. This is an **Integer**data type, **Primary key** and is defined as **Not Null**

*The****Primary key****is an attribute or set of attributes that can determine individual records in a table.*

*Defining an attribute****Not Null****will make sure there is a value given to the attribute (otherwise it will give an error).*

* Product\_ID – Id to identify the product that the customer bought. Also defined as Not Null
* Name – Name of a customer of **Text**type
* Gender – Gender of a customer of **Integer**type
* Age – Age of the customer of **Integer**type

*SQL keywords are case-insensitive so you can write the commands in UPPERCASE IF YOU WANT!*

Any SQL command can be executed using the **execute()** method of the **Cursor**object. You just need to write your query inside quotes and you may choose to include a**;** which is a requirement in some databases but not in SQLite. But it is always good practice so I will include it with my commands.

So, using the **execute()** method, we can create our table as shown here:

|  |  |
| --- | --- |
|  | # connect to existing database |
|  | conn = sqlite3.connect('.\sql\_db\Demo\_table.db') |
|  | cur = conn.cursor() |
|  |  |
|  | # create table in database |
|  | cur.execute('''CREATE TABLE CUSTOMER( |
|  | User\_ID INTEGER PRIMARY KEY NOT NULL, |
|  | Product\_ID INTEGER NOT NULL, |
|  | Name TEXT NOT NULL, |
|  | Gender TEXT NOT NULL, |
|  | AGE INTEGER NOT NULL, |
|  | CITY TEXT); |
|  | ''') |
|  |  |
|  | # commit and save changes to database |
|  | conn.commit() |

Perfect! Now that we have our table, let’s add some values to it.

# Inserting values in a SQL table

* **execute()**
* **executescript()**
* **executemany()**

A database table is of no use without values. So, we can use the **INSERT INTO**SQL command to add values to the table. The syntax for the command is as follows:

***INSERT INTO table\_name (column1, column2, column3, …)***

***VALUES (value1, value2, value3, …);***

But if we are adding values for all the columns in the table, we can just simplify things and get rid of the column names in the SQL statement:

***INSERT INTO table\_name***

***VALUES (value1, value2, value3, …);***

Like I said before, we can execute SQL statements using the **execute()** method. So let’s do that!

|  |  |
| --- | --- |
|  | cur.execute('''Insert Into Customer ('User\_ID','Product\_ID','Name','Gender','AGE','CITY') Values (1006, 3, 'Princess Diana', 'Female', 28, 'Amazons');''') |

What if we want to write multiple *Insert* commands in a single go? We could use the **executescript()**method instead:

|  |  |
| --- | --- |
|  | # Execute multiple commands at once |
|  | cur.executescript('''Insert Into CUSTOMER Values |
|  | (1005, 3, 'Clark Kent', 'Male', 36, 'Metropolis'); |
|  |  |
|  | Insert Into CUSTOMER Values |
|  | (1003, 4, 'Bruce Wayne', 'Male', 39, 'Gotham City'); |
|  |  |
|  | ''') |

Or just simply use the **executemany()** method without having to repeatedly write the *Insert Into* command every time! **executemany()**actually executes an SQL command using an iterator to yield the values:

|  |  |
| --- | --- |
|  | # Insert maultiple values into table at once |
|  | customers = [(1004, 2, 'John Wick', 'Male', 32, 'New York'), |
|  | (1001, 1, 'Tony Stark', 'Male', 35, 'New York'), |
|  | (1002, 3, 'Gordon Ramsey', 'Male', 38, 'London') |
|  | ] |
|  | cur.executemany('Insert Into CUSTOMER Values (?,?,?,?,?,?)', customers) |

These methods are not limited to the *Insert Into*command and can be used to execute any SQL statement.

Now that we have a few values in our table, let’s try to fetch those values from the database.

# Fetching Records from a SQL table

* **fetchone()**
* **fetchall()**

For fetching values from the database, we use the **SELECT**command and the attribute values we want to retrieve:

***SELECT column1, column2, … FROM table\_name;***

If you instead wanted to fetch values for all the attributes in the table, use the **\***character instead of the column names:

***SELECT \* FROM table\_name;***

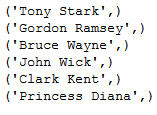
To fetch only a single record from the database, we can use the **fetchone()** method:

|  |  |
| --- | --- |
|  | # Fetch all rows of query result |
|  | cur.execute('SELECT \* FROM CUSTOMER;').fetchone() |

SQLite fetchone

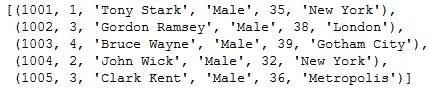
To fetch multiple rows, you can execute a SELECT statement and iterate over it directly using only a single call on the Cursor object:

|  |  |
| --- | --- |
|  | # iterate over the rows |
|  | for row in cur.execute('SELECT Name FROM CUSTOMER;'): |
|  | print(row) |



But a better way of retrieving multiple records would be to use the**fetchall()** method which returns all the records in a list format:

|  |  |
| --- | --- |
|  | # Fetch all rows of query result which returns a list |
|  | cur.execute('SELECT \* FROM CUSTOMER;').fetchall() |



Awesome! We now know how to insert values into a table and fetch those values. But since data scientists love working with Pandas dataframes, wouldn’t it be great to somehow load the values from the database directly into a dataframe?

Yes, there is and I am going to show you how to do that. But first, I am going to show you how to store your Pandas dataframe into a database, which is obviously a better way to store your data!

# Loading Pandas DataFrame into SQLite database

Pandas let us quickly write our data from a dataframe into a database using the**to\_sql()**method. The method takes the table name and Connection object as its arguments.

We use the dataframes from the Food Demand Forecasting hackathon on the DataHack platform which has three dataframes: order information, meal information, and center fulfillment information.

|  |  |  |
| --- | --- | --- |
|  |  | import pandas as pd |
|  |  | import sqlite3 |
|  |  |  |
|  |  | # read csv files |
|  |  | df\_train = pd.read\_csv('./train\_food/train.csv') |
|  |  | df\_meal = pd.read\_csv('./train\_food/meal\_info.csv') |
|  |  | df\_center = pd.read\_csv('./train\_food/fulfilment\_center\_info.csv') |
|  |  |  |
|  |  | # connect to database |
|  |  | conn = sqlite3.connect('.\sql\_db\FOOD.db') |
|  |  | cur = conn.cursor() |
|  |  |  |
|  |  | # load dataframes into database |
|  |  | df\_train.to\_sql("train", conn) |
|  |  | df\_meal.to\_sql('meal',conn) |
|  |  | df\_center.to\_sql('centers',conn) |

We now have three tables in the database for each dataframe. It is easy to check them using the **read\_sql\_query()**method which we will explore in the next section where we will see how to load a database into a Pandas dataframe.

# Reading an SQLite Database into a Pandas DataFrame

The**read\_sql\_query()**method of the Pandas library returns a DataFrame corresponding to the result of an SQL query. It takes as an argument the query and the Connection object to the database.

We can check the values in the tables using the real\_sql\_query() method:

|  |  |
| --- | --- |
|  | df = pd.read\_sql\_query("select \* from meal;", conn) |
|  | df.head() |

|  |  |
| --- | --- |
|  | df = pd.read\_sql\_query("select \* from centers;", conn) |
|  | df.head() |

|  |  |
| --- | --- |
|  | df = pd.read\_sql\_query("select \* from train;", conn) |
|  | df.head() |

Perfect! Now let’s try to run some queries on these tables and understand a few important SQL commands that will come in handy when we try to analyze data from the database.

# Querying our SQLite Database

**Where clause**

The first important clause is the **WHERE**clause. It is used to filter the records based on a condition:

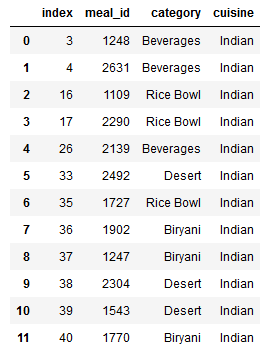
***SELECT column1, column2, … FROM table\_name***

***WHERE condition;***

*We can always use the \* character if we want to retrieve values for all the columns in the table.*

We can use it to query and retrieve only the Indian cuisine meals from the **meal** table:

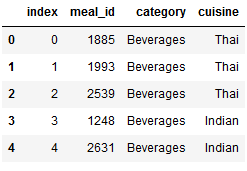
|  |  |
| --- | --- |
|  | # WHERE clause |
|  | df = pd.read\_sql\_query('''Select \* from meal |
|  | Where cuisine='Indian'; ''', conn) |
|  | df |



Here, we have retrieved all the 12 records that matched our given condition. But what if we only wanted to retrieve the top 5 records that satisfy our condition? Well, we could use the **LIMIT**clause in that case.

**LIMIT**clause returns only the specified number of records and is useful when there are a large number of records in the table.

|  |  |
| --- | --- |
|  | # LIMIT clause |
|  | df = pd.read\_sql\_query('''Select \* from meal |
|  | Where category='Beverages' |
|  | Limit 5;''', conn) |
|  | df |



Here, we returned only the top 5 records from those that matched our given condition.

# Group By statement

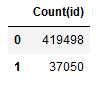
**GROUP BY** statement separates rows into different groups based on an attribute and can be used to apply an **aggregate function (COUNT, MIN, MAX, SUM)**on the resultant groups:

***SELECT column1, column2, … FROM table\_name***

***GROUP BY column\_name;***

We can use the GROUP BY statement to compare the number of orders for meals that received email promotions to those that did not. We will group the records on the **emailer\_for\_promotion** column and apply the **COUNT aggregate function** on the **id** column since it contains unique values. This will return the total number of rows belonging to each group:

|  |  |
| --- | --- |
|  | # GROUPBY statement |
|  | df = pd.read\_sql\_query('''Select Count(id) from train |
|  | Group by emailer\_for\_promotion;''',conn) |
|  | df |



Here we can see that there were more orders for meals that did not have an email promotion. But if we want to order our result, we can use the ORDER BY statement.

# Order By clause

**ORDER BY** clause is used to sort the result into ascending or descending order using the keywords **ASC** or **DESC** respectively. By default, it sorts the records in ascending order:

***SELECT column1, column2, … FROM table\_name***

***ORDER BY column1, column2, … ASC|DESC;***

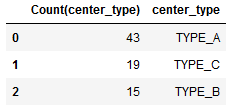
|  |  |
| --- | --- |
|  | # ORDER BY statement |
|  | df = pd.read\_sql\_query('''Select count(id) as Email from train |
|  | Group by emailer\_for\_promotion |
|  | Order by Count(id);''',conn) |
|  | df |



Here I have combined two clauses: Group By and Order By. The Group By clause groups the values based on the *email\_for\_promotion*attribute and the Order By attribute orders output based on the count of the rows in each group. *We can combine a bunch of clauses to extract more precise information from the database.*

To sort the result in descending order, just type in the keyword **DESC**:

|  |  |
| --- | --- |
|  | # ORDER BY descending |
|  | df = pd.read\_sql\_query('''Select Count(center\_type), center\_type |
|  | From centers |
|  | Group By center\_type |
|  | Order by Count(center\_type) Desc; ''',conn) |
|  | df |



# Having clause

The **HAVING**clause is used to query on the results of another query run on the database. It applies a filter on the groups returned by a previous query. It should not be confused with the WHERE clause that applies the filter condition before grouping.

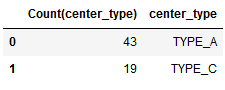
HAVING is used to filter records after grouping. Hence, the HAVING clause is always used after the GROUP BY statement:

***SELECT column1, column2, … FROM table\_name***

***GROUP BY column\_name***

***HAVING condition;***

|  |  |
| --- | --- |
|  | # HAVING clause |
|  | df = pd.read\_sql\_query('''Select Count(center\_type), center\_type |
|  | From centers |
|  | Group By center\_type |
|  | Having Count(center\_type) > 15; ''', conn) |
|  | df.head() |



Here, we returned only those groups that had a count of more than 15.

# Join clause

**Join**clause is a very interesting and important SQL clause. It retrieves and combines data from multiple tables on the same query based on a common attribute:

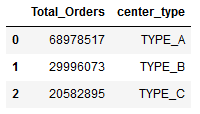
***SELECT column1, column2, … FROM table1***

***INNER JOIN table2***

***ON table1.column\_name= table2.column\_name;***

In our database, we can retrieve data from the *centers*and *train*tables since they share the common attribute **center\_id**:

|  |  |
| --- | --- |
|  | # JOIN clause |
|  | df = pd.read\_sql\_query('''Select Sum(train.num\_orders) as "Total\_Orders", centers.center\_type from train |
|  | Inner Join centers |
|  | On train.center\_id = centers.center\_id |
|  | Group By centers.center\_type;''', conn) |
|  | df.head() |



The **INNER JOIN** clause combines the two tables, *train*and *centers*, on the common attribute *center\_id*specified by the statement **train.center\_id = centers.center\_id**. This means records having the same *center\_id*in both the columns will concatenate horizontally.

This way we were able to retrieve the *center\_type*, from *centers*, and the corresponding total number of orders from the *train*table. The **.**operator is very important here as it lets the database know which table the column belongs to.

# Update statement

Now, let’s say there was a glitch in the system and the *base price* for all the orders was saved as 10 more than the actual amount. We want to make that update in the database as soon as we found the mistake.

In such a situation, we will use the **UPDATE**SQL command.

The UPDATE command is used to modify existing records in a table. However, always make you sure you provide which records need to be updated in the WHERE clause otherwise all the records will be updated!

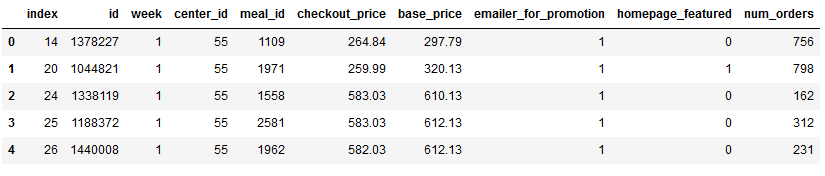
***UPDATE table\_name***

***SET column1 = value1, column2 = value2, …***

***WHERE condition;***

Let’s have a look at the table before the update:

|  |  |
| --- | --- |
|  | df = pd.read\_sql\_query('''Select \* from train |
|  | Where emailer\_for\_promotion = 1; ''', conn) |
|  | df.head() |

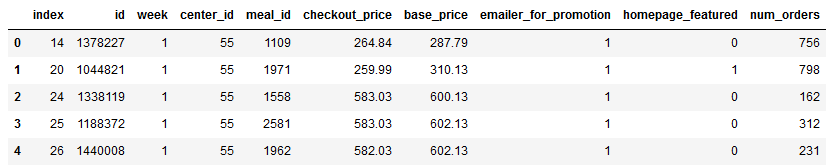


Decrease all the base prices by 10 for orders containing meals that had an email promotion:

|  |  |
| --- | --- |
|  | # UPDATE statement |
|  | conn.execute('''Update train |
|  | Set base\_price = base\_price - 10 |
|  | Where emailer\_for\_promotion = 1; ''') |

Finally, here’s a look at the updated table:

|  |  |
| --- | --- |
|  | df = pd.read\_sql\_query('''Select \* from train |
|  | Where emailer\_for\_promotion = 1; ''', conn) |
|  | df.head() |



All the records have been correctly updated!