

## What is SQL?

---

SQL, which stands for Structured Query Language, is a way to interact with relational databases and tables in a way that allows us humans to glean specific, meaningful information.

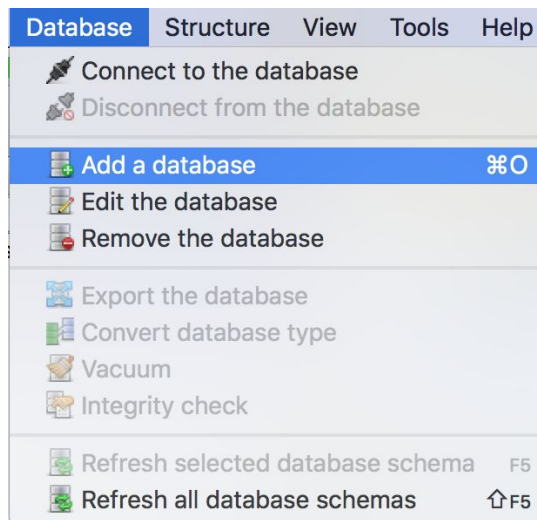
There are many “flavors” of SQL, but for our mystery we’ll be using [SQLite](#), which you can download [here](#). We also recommend downloading [SQLiteStudio](#), which is a good graphical interface to use to inspect your data and write queries.

## What is a SQLite database file and how to import it?

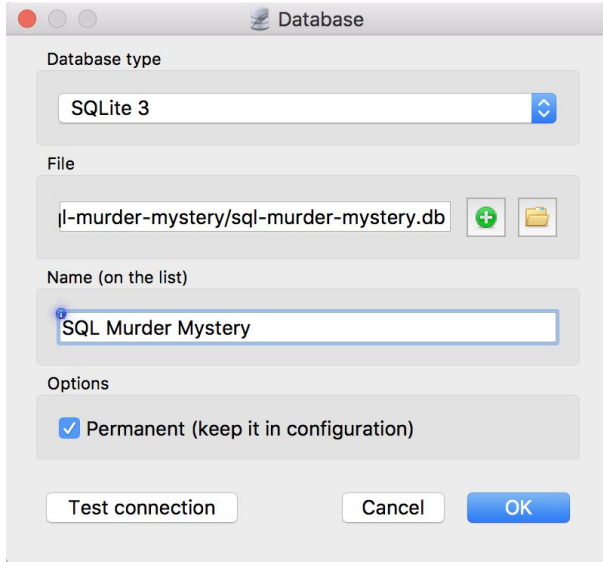
---

In SQLite, a .db file, otherwise known as a database file, is a collection of tables, which are exactly like the tables you might already have dealt with in the past in programs such as Microsoft Excel or Google Sheets. Within a table, each row records a data point and each column contains a specific type of data.

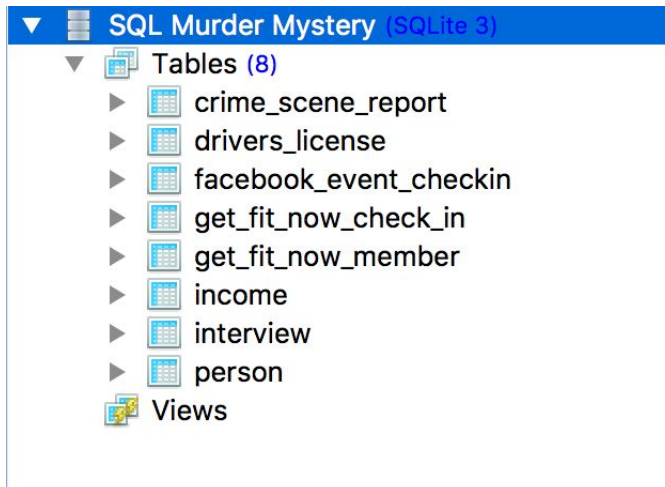
To load in our database file, open SQLiteStudio, click “Database” and then “Add a database”:



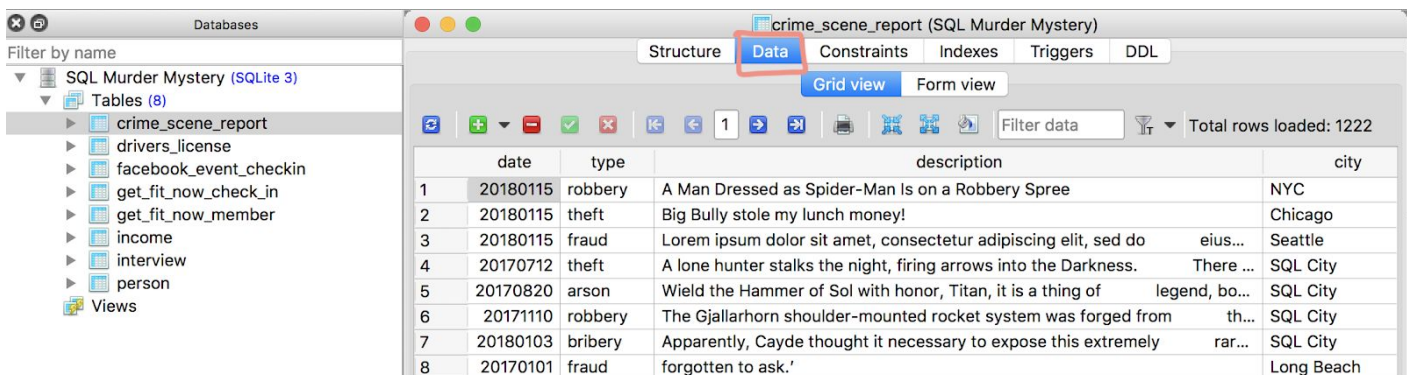
Locate the sql-murder-mystery.db file, give it a name and hit “OK”:



Double click on the database to see its content:



Double click on a table and click "Data" to inspect a table:



## What is an ERD?

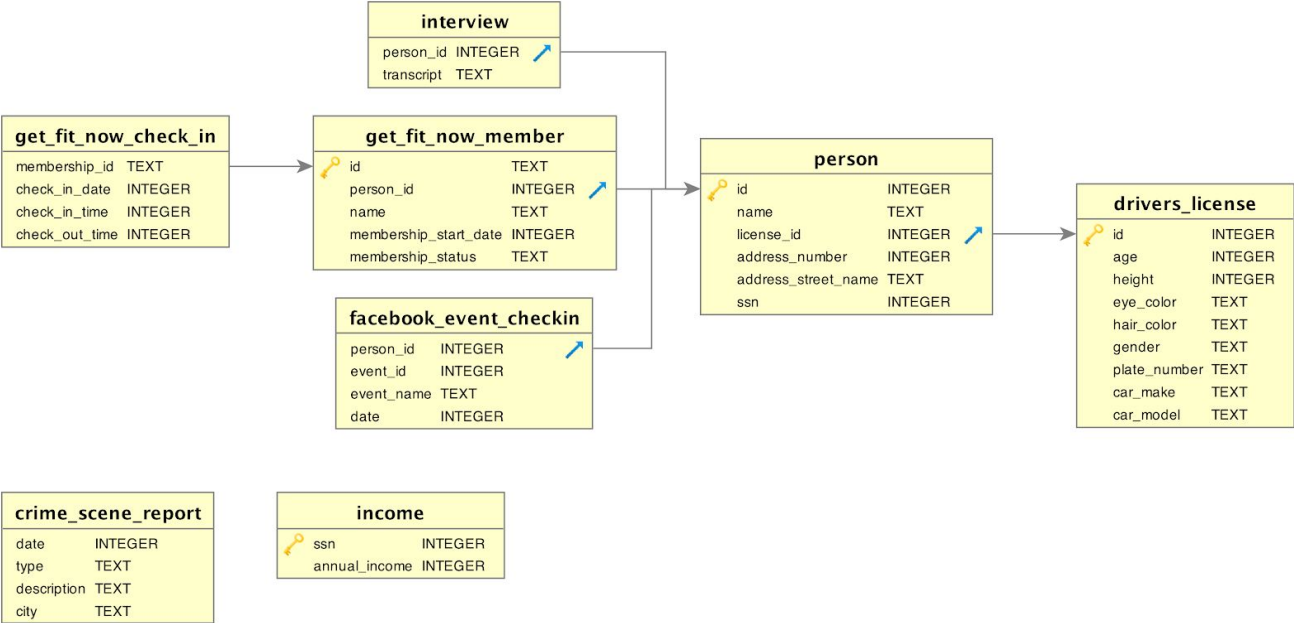
ERD, which stands for Entity Relationship Diagram, is a visual representation of the relationships among all relevant tables within a database. You can find the ERD for our SQL Murder Mystery database below. The diagram shows that each table has a name (top of the box, in bold), a list of column names (on the left) and their corresponding data types (on the right, in all caps), all of which should correctly match what you see in SQLiteStudio.

There are also some gold key icons, blue arrow icons and gray arrows on the ERD. A gold key indicates that the column is the **primary key** of the corresponding table, and a blue arrow indicates that the column is the **foreign key** of the corresponding table.

**Primary Key:** a unique identifier for each row in a table.

**Foreign Key:** used to reference data in one table to those in another table.

If two tables are related, the matching columns, i.e. the common identifiers of the two tables, are connected by a gray arrow in the diagram.



(Powered by [DbVisualizer](#))

## What is a query?



If you take a look at the tables you just imported in SQLiteStudio, you will see that the tables are huge! There are so many data points, and it simply isn't possible to go

through the tables row by row to find the information we need. What are we supposed to do?

This is where queries come in. Queries are statements we construct to grab specific rows of table(s) that match a set of criteria. Queries read like natural English (for the most part). For example:

```
SELECT name FROM person;
```

The query above **SELECTs** the `name` column **FROM** the `person` table, and the semicolon (;) indicates that it is the end of the query.

In SQLiteStudio, click this () icon to open SQL editor, write your queries in the top field and click this () icon to run the queries.

## What elements does a SQL query have?

---

A SQL query can contain:

- SQL keywords (like the **SELECT** and **FROM** above),
- Column names (like the `name` column above),
- Table names (like the `person` table above),
- Wildcards,
- Functions,
- Specific filtering criteria,
- Etc.

## SQL Keywords and Wildcards

---

SQL keywords are used to specify actions in your queries. SQL keywords are not case sensitive, but we suggest using all caps for SQL keywords so that you can easily set them apart from the rest of the query. Some frequently used keywords are:

### **SELECT**

**SELECT** allows us to grab columns from the database:

- \* (asterisk): it is used after **SELECT** to grab all columns from the table;

- **column\_name(s)**: to select specific columns, put the names of the columns after **SELECT** and use commas to separate them.

### **FROM**

**FROM** allows us to specify which table(s) we care about; to select multiple tables, list the table names and use commas to separate them.

### **WHERE**

The **WHERE** clause in a query is used to filter results by specific criteria. For example:

```
SELECT * FROM person WHERE name = "Kinsey Erickson";
```

The query above **SELECTS** all columns (**\***) **FROM** the **person** table **WHERE** the name of the person is "Kinsey Erickson".

The **AND** keyword is used to string together multiple filtering criteria so that the filtered results meet each and every one of the criterion. For example:

```
SELECT * FROM crime_scene_report WHERE type = "theft" AND city = "Chicago";
```

The above query will select all the thefts that took place in Chicago from the **crime\_scene\_report** table.

If you only need the filtered results to satisfy at least one of the criteria, use **OR** to string the criteria together. For example:

```
SELECT * FROM crime_scene_report WHERE city = "Seattle" OR city = "Baltimore";
```

The above query will select all the crimes that took place in either Seattle or Baltimore from the **crime\_scene\_report** table.

In the **WHERE** clause, there are 3 commonly used operators:

- **=**  
The equal sign (**=**) indicates that results have to match the exact value specified.

- **BETWEEN ... AND ...**

The **BETWEEN** operator and the **AND** keyword is used to specify a range.

```
SELECT * FROM person WHERE address_street_name = "Northwestern Dr" AND address_number BETWEEN 100 AND 1000;
```

The query above will select all the columns for people from the `person` table whose street name is “Northwestern Dr” and address number is **BETWEEN** 100 **AND** 1000.

- **LIKE**

To search for a pattern, use the **LIKE** operator with either the percentage (%) and/or the underscore ( \_ ) wildcards, which act as placeholders:

```
SELECT * FROM person WHERE address_street_name LIKE "Northwestern%";
```

The query above will select all the columns for people from the `person` table whose street name starts with the word “Northwestern”. The table below from [w3schools.com](https://www.w3schools.com) explains how the **LIKE** operator works with % and \_ wildcards in different circumstances:

Here are some examples showing different LIKE operators with '%' and '\_' wildcards:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"
WHERE CustomerName LIKE '%a'	Finds any values that ends with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that starts with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

(source: [https://www.w3schools.com/sql/sql\\_like.asp](https://www.w3schools.com/sql/sql_like.asp))

## SQL Aggregate Functions

Sometimes the questions you want to ask aren't as simple as finding the row of data that fits a set of criteria. You may want to ask more complex questions such as “Who is the oldest person?” or “Who is the shortest person?” Aggregate functions can help you answer these questions.

Try running the following query:

```
SELECT id, age FROM drivers_license;
```

The results are what you'd expect - a list of ids and the corresponding ages of people listed in the `drivers_license` table.

Now try:

```
SELECT id, MAX(age) FROM drivers_license;
```

You should get a single row as an output, which will be the row containing the maximum age found in the table. Here are some other aggregate functions you can try:

- **MIN:** finds the minimum value
- **SUM:** calculates the sum of the specified column values
- **AVG:** calculates the average of the specified column values
- **COUNT:** counts the number of specified column values

There are a few other helpful keywords that often are used in conjunction with aggregate functions. Try the following query:

```
SELECT id, height FROM drivers_license ORDER BY height;
```

You'll notice that the output was ordered, starting with the shortest. You can also go highest to lowest:

```
SELECT id, height FROM drivers_license ORDER BY height DESC;
```

Now try this query:

```
SELECT age, COUNT(age) FROM drivers_license GROUP BY age;
```

Here, we've used the **GROUP BY** keyword to show us how many people of each age exist in the table. You can even have the above query show you the results from oldest to youngest by adding on the **ORDER BY height DESC** command at the end, just as we did in a previous query.

## SQL JOINS

---

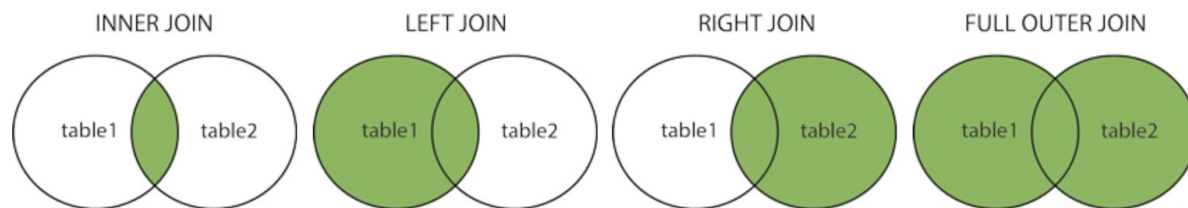
Until now, we've been asking questions that can be answered by considering data from only a single table. But what if we need to ask more complex questions that simultaneously require data from two different tables? That's where **JOIN** comes in.

The graph below from [w3schools.com](http://w3schools.com) illustrates and explains 4 types of **JOINS**:

# Different Types of SQL JOINS

Here are the different types of the JOINS in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table



(source: [https://www.w3schools.com/sql/sql\\_join.asp](https://www.w3schools.com/sql/sql_join.asp))

Here's an example of **INNER JOIN**:

```
SELECT person.name, drivers_license.age FROM drivers_license
JOIN person ON drivers_license.id = person.license_id;
```

If you run this query in SQLiteStudio, you should see a table with 2 columns, name and age.

Let's break it down:

Step 1:

```
SELECT person.name, drivers_license.age
```

Just like the queries we wrote before, put the names of the columns you want to see after the **SELECT** keyword. Since there are more than one table, make sure to specify the table name for each column name, in `table_name.column_name` format. In the example above, we want to see the `name` column from the `person` table and the `age` column from the `drivers_license` table.

Step 2:

```
SELECT person.name, drivers_license.age FROM drivers_license
JOIN person
```

Recall that the **FROM** keyword is used to specify which tables you want the information from. There are four types of JOIN keywords we can use to join two tables together, as shown in the graph above: **(INNER) JOIN** / **LEFT (OUTER) JOIN** / **RIGHT (OUTER) JOIN** / **FULL (OUTER) JOIN**. Note that the words in the parentheses are optional. In the example above, we are performing an **(INNER) JOIN** of the `drivers_license` table and the `person` table.



Step 3:

```
SELECT person.name, drivers_license.age FROM drivers_license
JOIN person ON drivers_license.id = person.license_id;
```

In order for two tables to be joined together, they must contain matching columns with common identifiers. From the ERD diagram, we know that the `id` column in `drivers_license` table is “connected” to the `license_id` column in the `person` table. After the `ON` keyword, we always specify through which matching columns can the two tables be joined together.

Step 4:

```
SELECT person.name, drivers_license.age FROM drivers_license
JOIN person ON drivers_license.id = person.license_id WHERE ...;
```

Write the filtering criteria as you normally would in the `WHERE` clause.

In sum, the general structure of a `JOIN` query is:

```
SELECT table_name_1.column_name, ... table_name_2.column_name FROM
table_name_1 (INNER) JOIN / LEFT (OUTER) JOIN / RIGHT (OUTER)
JOIN / FULL (OUTER) JOIN table_name_2 ON
table_name_1.matching_column_1 = table_name_2.matching_column_2;
```

A bit confused? Check out more examples on [Tutorialspoint](#) or [W3Schools](#).