

---

# **A Gentle Introduction to SQL Documentation**

*Release 0.0.1*

**Troy Thibodeaux and Serdar Tumgoren**

**Nov 05, 2019**



|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Part I</b>   | <b>3</b>  |
| 1.1      | Hello SQL! . . . . .  | 3         |
| 1.2      | Installing DB Browser for SQLite . . . . .                              | 4         |
| 1.3      | Creating the First Database . . . . .                                   | 5         |
| 1.4      | Creating a Table . . . . .  | 5         |
| 1.5      | Inserting Data . . . . .  | 8         |
| 1.6      | Querying Data . . . . .   | 9         |
| 1.7      | Using DISTINCT to get a distinct set . . . . .                          | 10        |
| 1.8      | The WHERE Clause . . . . .  | 12        |
| 1.9      | Combining conditions with AND and OR . . . . .                          | 13        |
| 1.10     | Sorting results with ORDER BY . . . . .                                 | 15        |
| 1.11     | Changing values with UPDATE . . . . .                                   | 17        |
| 1.12     | Deleting data . . . . .   | 19        |
| 1.13     | Further References . . . . .  | 20        |
| <b>2</b> | <b>Part II</b>  | <b>21</b> |
| 2.1      | Getting started . . . . .   | 21        |
| 2.2      | Nothing can come of nothing: Using IS NULL . . . . .                    | 21        |
| 2.3      | Knowing your limitations: Using LIMIT . . . . .                         | 23        |
| 2.4      | Casting a wider net with LIKE . . . . .                                 | 24        |
| 2.5      | Using string functions: SUBSTR(), TRIM(), UPPER(), LOWER() . . . . .    | 27        |
| 2.6      | Pull yourself together: The concatenate operator (  ) . . . . .         | 30        |
| 2.7      | Pick One: Using BETWEEN and IN (NOT IN) . . . . .                       | 31        |
| 2.8      | Aggregate Functions: COUNT, MAX, MIN, SUM, AVG . . . . .                | 32        |
| 2.9      | Beyond functions: Custom calculations . . . . .                         | 34        |
| 2.10     | Subqueries, the Russian dolls of SQL . . . . .                          | 36        |
| 2.11     | GROUP BY . . . . .  | 37        |
| 2.12     | HAVING . . . . .  | 39        |
| 2.13     | Revisiting subqueries . . . . .   | 42        |
| 2.14     | Conclusion . . . . .  | 44        |
| <b>3</b> | <b>Part III</b>   | <b>45</b> |
| 3.1      | Spreading the data around: Data Normalization . . . . .                 | 45        |
| 3.2      | Referentially speaking: Associating tables using foreign keys . . . . . | 47        |
| 3.3      | Reaching across the aisle using JOIN . . . . .                          | 49        |
| 3.4      | Explicit JOIN syntax . . . . .  | 50        |
| 3.5      | OUTER JOIN . . . . .  | 52        |

|          |   |           |
|----------|---|-----------|
| 3.6      | Why be normal? Denormalization as an informed choice. . . . . | 53        |
| 3.7      | Conclusion . . . . .  | 54        |
| 3.8      | Further Resources . . . . .                                   | 54        |
| <b>4</b> | <b>Appendix</b>   | <b>57</b> |
| 4.1      | Importing data from a file . . . . .                          | 57        |
| 4.2      | Saving scripts . . . . .                                      | 59        |
| <b>5</b> | <b>Indices and tables</b>                                     | <b>61</b> |

This tutorial was crafted by [Troy Thibodeaux](#) as a human-friendly introduction to the world of databases and SQL. It introduces database skills from the ground up using SQLite and a small set of data from the world of campaign finance.

This tutorial largely hews to Troy's original [SQL-Tutorial](#), but updates the material to work with *DB Browser for SQLite*. Source code for this tutorial lives on [Github](#).



### 1.1 Hello SQL!

SQL or Structured Query language is the language used to communicate with relational databases. What are relational databases? Well, most of the popular database systems you may know, such as MS Access, MySQL or SQLite, are all relational. That is, they all use a relational model, which, it turns out, can be described much like a spreadsheet:

- Data are organized into tables (relations) that represent a collection of similar objects (e.g. contributors).
- The columns of the table represent the attributes that members of the collection share (last name, home address, amount of contribution).
- Each row in the table represents an individual member of the collection (one contributor).
- And the values in the row represent the attributes of that individual (Smith, 1228 Laurel St., \$250).

Much of the power of a relational database lies in the ability to query these relations, both within a table (give me all contributors who donated at least \$500 and who live in Wyoming) and among tables (from the contributors, judges and litigants tables, give me all contributors who donated at least \$1000 to Judge Crawford and who also had legal cases over which Judge Crawford presided). SQL is the powerful and rather minimalist language we use to ask such questions of our data in a relational database. How minimalist is SQL? The basic vocabulary for querying data comes down to a few main verbs:

```
SELECT
INSERT
UPDATE
DELETE
```

I imagine you can guess what each of those verbs does, even if you've never written a database query.

To create and change the structure of tables in the database, there are a few other verbs to use:

```
CREATE  
DROP  
ALTER
```

Those are the keywords that perform almost everything you need to do. The language also includes a number of modifiers that help specify the action of the verbs, but the core list comes down to a couple dozen words. These basic keywords are common across pretty much all relational databases. A specific database management system (Access, MySQL or SQLite) may add its own extensions to the common keywords, but the lion's share of the work is done with this handful of words, and they're basically the same across database applications.

By combining these simple keywords, you can create remarkably complex and specific queries. And the basic syntax still reads fairly clearly:

```
SELECT last_name FROM contributors WHERE state = 'WY';
```

The SQL query above reads pretty much like the English sentence for the same request:

```
Select the last name from the contributors table where the contributor's state is WY.
```

If you're using a graphical interface such as a datagrid, that interface is simply constructing queries like these behind the scenes. So, why not take command of your queries and write them yourself?

A couple of things off the bat:

- SQL keywords are not case-sensitive. So capitalizing SELECT in the statement above is optional. Using all caps for keywords is considered good form, though, because it helps distinguish keywords from table names or other non-keywords.
- The statement ends with a semi-colon. This is the standard way of ending a statement in SQL. Some systems enforce this convention.

So, let's dive in. For this tutorial, we will be using SQLite, a free and open source database manager that's lightweight and portable.

## 1.2 Installing DB Browser for SQLite

To create our own databases, we'll use the free and open source [DB Browser for SQLite](#). Per their documentation:

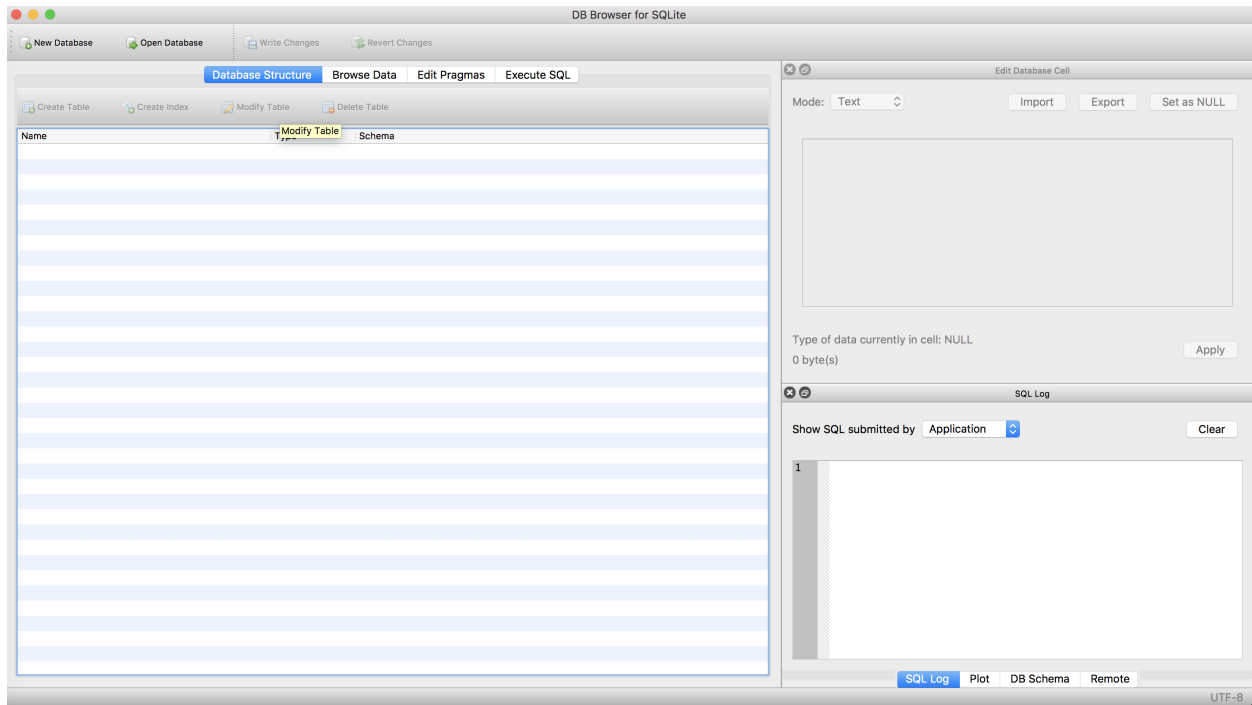
DB Browser for SQLite is a high quality, visual, open source tool to create, design, and edit database files compatible with SQLite. It is for users and developers wanting to create databases, search, and edit data. It uses a familiar spreadsheet-like interface, and you don't need to learn complicated SQL commands.



Go to [this page](#), download the installer appropriate for your machine, and execute the installer.

Once finished, search for the program and fire it up.

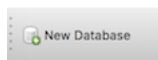
On a Mac, you can hit Command + Space bar and type your search to find the program, or search in **Launchpad**.



## 1.3 Creating the First Database

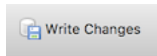
Mousing over each of the icons at the top of the *DB Browser* tool will show what the icon does.

To create a database, simply click on the icon for “New Database”:



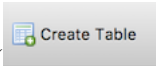
Name the database “contributors” and save it anywhere you like (the desktop will work, or your documents folder). This single file will contain the entire database you create.

**Note:** As you do work in *DB Browser*, be aware that it will not automatically save your work. If you plan to step away from the tutorial, be sure to save your changes by clicking the “Write Changes” button:



You can also *save SQL queries* as individual scripts.

## 1.4 Creating a Table

Click the “Create Table” icon () , and you’ll get a form allowing you to create a new table.

To create a table, we have to define the attributes or columns that make it up. For each column, we define the datatype of the data it will contain.

Name the table “contributors” and begin creating columns as below by clicking the “Add field” button.

**Note:** You should name and order the fields AND fill in the drop-down menu and checkboxes exactly as displayed!

**Edit table definition**

Table

**contributors**

▼ Advanced

Fields

Add field
 Remove field
 Move field up
 Move field down

| Name       | Type    | Not                                 | PK                                  | AI                                  | U                        | Default | Check |
|------------|---------|-------------------------------------|-------------------------------------|-------------------------------------|--------------------------|---------|-------|
| id         | INTEGER | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input type="checkbox"/> |         |       |
| last_name  | TEXT    | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/> |         |       |
| first_name | TEXT    | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/> |         |       |
| city       | TEXT    | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/> |         |       |
| state      | TEXT    | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/> |         |       |
| zip        | TEXT    | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/> |         |       |
| amount     | INTEGER | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/> |         |       |

```

1 CREATE TABLE `contributors` (
2   `id` INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
3   `last_name` TEXT,
4   `first_name` TEXT,
5   `city` TEXT,
6   `state` TEXT,
7   `zip` TEXT,
8   `amount` INTEGER
9 );
  
```

Cancel OK

### Important notes

Some important things to note:

- As you start adding fields and options, note how the table creation SQL in the bottom pane dynamically updates.
- The `id` field will be a unique identifier for each contributor (and therefore will be the “Primary Key” for the row), which is why we checked the `PK` box for this one field. Checking the `AI` box will make this integer automatically increment for each row we add (so each new row will have a new `id`). Finally, this field should not be null or empty (because we need it as the unique identifier), so we check the `Not` box as well.
- The next five columns will all contain text strings of undetermined lengths (last names, for example, come in all kinds of lengths), so we’ll use the `TEXT` datatype, which allows for text of varying length.

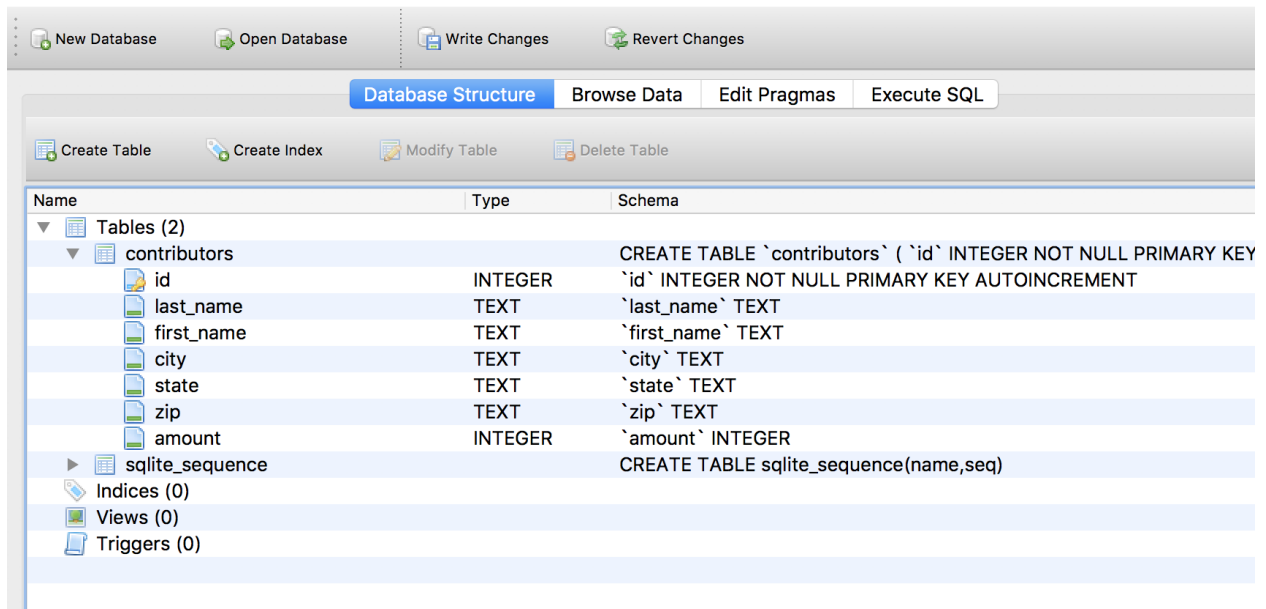
By the way, it may seem strange that the `zip` column uses a `TEXT` datatype, but remember that some zips start with a 0 (00501 is in NY). So, we want to treat this column as a string of text, rather than as a number (which would be 501).

Click **OK** and *DB Browser* will create the table based on your specifications, by executing the full SQL statement in the lower pane of the table creation window:

```
CREATE TABLE `contributors` (
  `id`      INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
  `last_name` TEXT,
  `first_name` TEXT,
  `city`    TEXT,
  `state`   TEXT,
  `zip`     TEXT,
  `amount`  INTEGER
);
```

The syntax should be fairly clear, since it just reflects the choices we made in the form. It’s creating a table called “contributors” with the fields and data types we’ve defined.

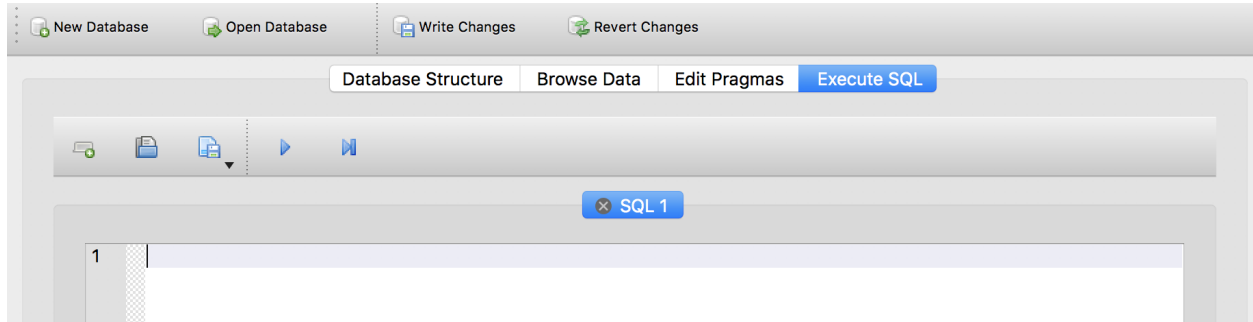
You should now have a “contributors” table in the list in the Database Structure panel of the manager. Clicking the arrow beside the listing for “contributors” will show you the column list for the table.



## 1.5 Inserting Data

Now that we have a table in the database, we can start inserting data. This task is accomplished with (oddly enough) an `INSERT` statement.

Click the “Execute SQL” tab in the second menu row, under the icons for creating/opening databases:



The top pane with the flashing cursor is where you can write SQL queries. Since we don’t have data in the table yet, let’s go ahead and insert some by copying and pasting the below SQL into the pane with the flashing cursor.

```
INSERT INTO contributors (last_name, first_name, city, state, zip, amount)
VALUES ('Buffet', 'Warren', 'Omaha', 'Nebraska', '68101', 1500);
```

This is a little more obscure than the `CREATE` or `SELECT` syntax, but it’s still fairly clear. To insert a row in the table, we execute the `INSERT INTO` statement with a table name, a list of columns to populate, and the `VALUES` for each of those columns. **The order of the columns in the column list must match the order of values in the values list.**

It’s very important to surround text values with single quotation marks. Two things to note:

- The quotation marks indicate to SQL that this is a literal string (the word ‘Buffet’), rather than a column name or other special usage.
- SQL uses single quotation marks around text strings. Some database systems will also accept double quotes, but some will throw an error.
- The commas between values are placed outside of the quote marks, not inside.

Notice that we didn’t insert a value for `id`. Because we set that column to `AUTOINCREMENT`, SQLite will populate the `id` with the next integer in the sequence. So, we don’t need to worry about choosing unique ids; SQLite takes care of it.

Finally, we didn’t include dollar signs or commas in the “amount” column. We created the “amount” column as an integer, so we should only insert integers there. (Different database management systems will react differently if you try to insert non-numeric characters in an integer column; it’s best to avoid doing so.)

If you haven’t done so already, click the Execute SQL button .

The bottom pane should say “Query executed successfully” followed by a copy of the SQL that was executed. **Success! You’ve added data!**

You can view the data by going to the “Browse Data” tab:

The screenshot shows a database management tool with tabs for 'Database Structure', 'Browse Data', 'Edit Pragmas', and 'Execute SQL'. The 'Browse Data' tab is active, showing a table named 'contributors'. The table has columns: id, last\_name, first\_name, city, state, zip, and amount. A single record is displayed with id 1, last\_name Buffet, first\_name Warren, city Omaha, state Nebraska, zip 68101, and amount 1500. There are buttons for 'New Record' and 'Delete Record'.

| id | last_name | first_name | city  | state    | zip   | amount |
|----|-----------|------------|-------|----------|-------|--------|
| 1  | Buffet    | Warren     | Omaha | Nebraska | 68101 | 1500   |

Just so we'll have some data to play with, let's execute a few more `INSERT` statements. Go back to the "Execute SQL" tab and paste in these lines:

```
INSERT INTO contributors (last_name, first_name, city, state, zip, amount) VALUES (
↳ 'Winfrey', 'Oprah', 'Chicago', 'IL', '60601', 500);
INSERT INTO contributors (last_name, first_name, city, state, zip, amount) VALUES (
↳ 'Chambers', 'Anne Cox', 'Atlanta', 'GA', '30301', 200);
INSERT INTO contributors (last_name, first_name, city, state, zip, amount) VALUES (
↳ 'Cathy', 'S. Truett', 'Atlanta', 'GA', '30301', 1200);
```

You can paste all three lines into the SQL text box at the same time. The semi-colons indicate the end of each statement.

Before inserting these new records, you should delete the original `INSERT` statement to avoid re-running it, which would result in a duplicate record.

Click the "Execute SQL" button.

You can view the new records in the "Browse Data" tab. **You should see 4 rows in total now.**

## 1.6 Querying Data

Now that we have a small data set to use, let's start querying it by using the `SELECT` statement.

Navigate to the "Execute SQL" panel and type the following into the SQL text box:

```
SELECT * FROM contributors;
```

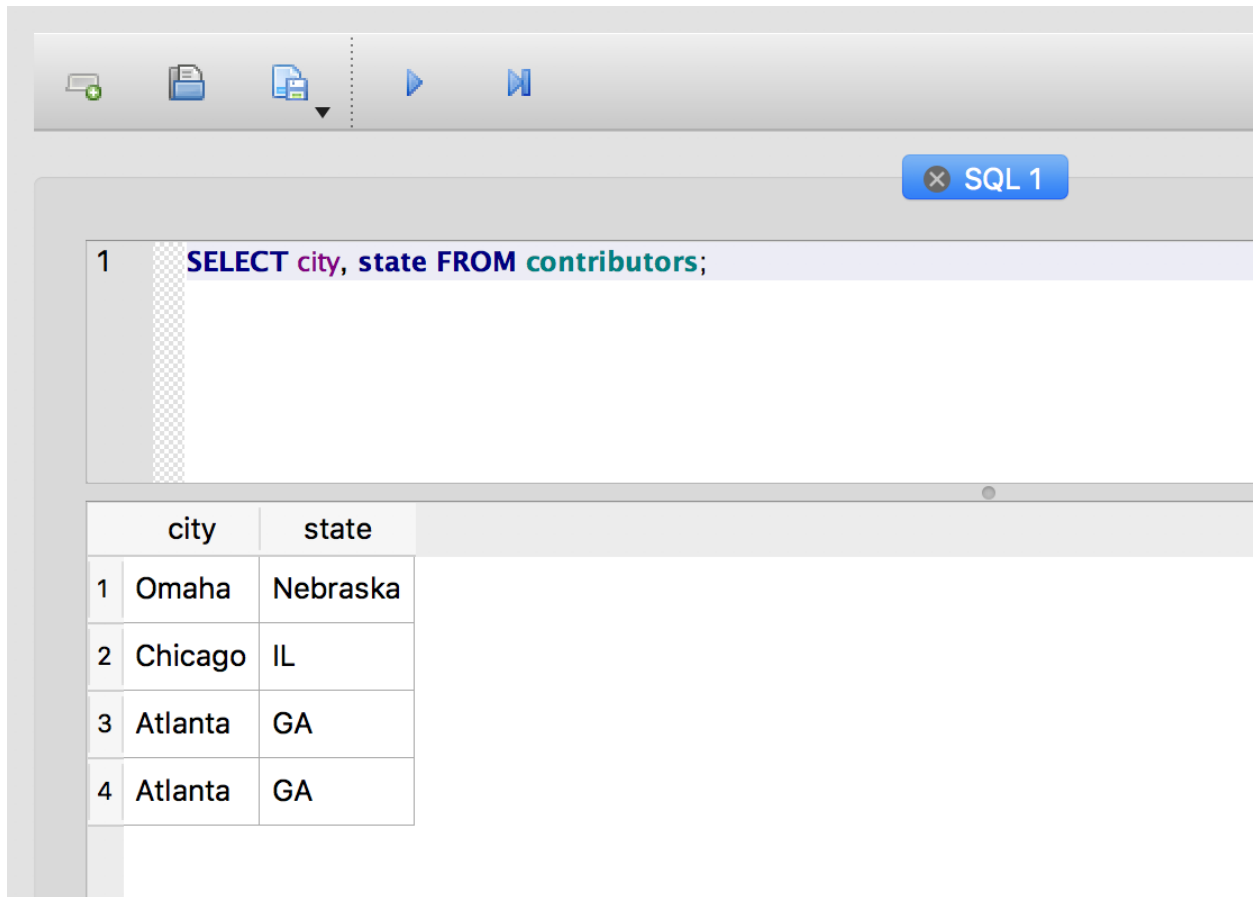
Now click the "Execute SQL" button.

You should see a nice grid display of all contributors you've added. The `*` character is a common wildcard. In this `SELECT` statement, it is used to retrieve all columns. So, we have selected all columns from all rows in the contributors table.

To define which columns of data you want to return, simply provide a comma-separated list of column names to `SELECT`:

```
SELECT city, state FROM contributors;
```

Clicking "Execute SQL" should give you a two-column table of cities and states.



The screenshot shows a SQL editor window titled "SQL 1". The query entered is:

```
1 SELECT city, state FROM contributors;
```

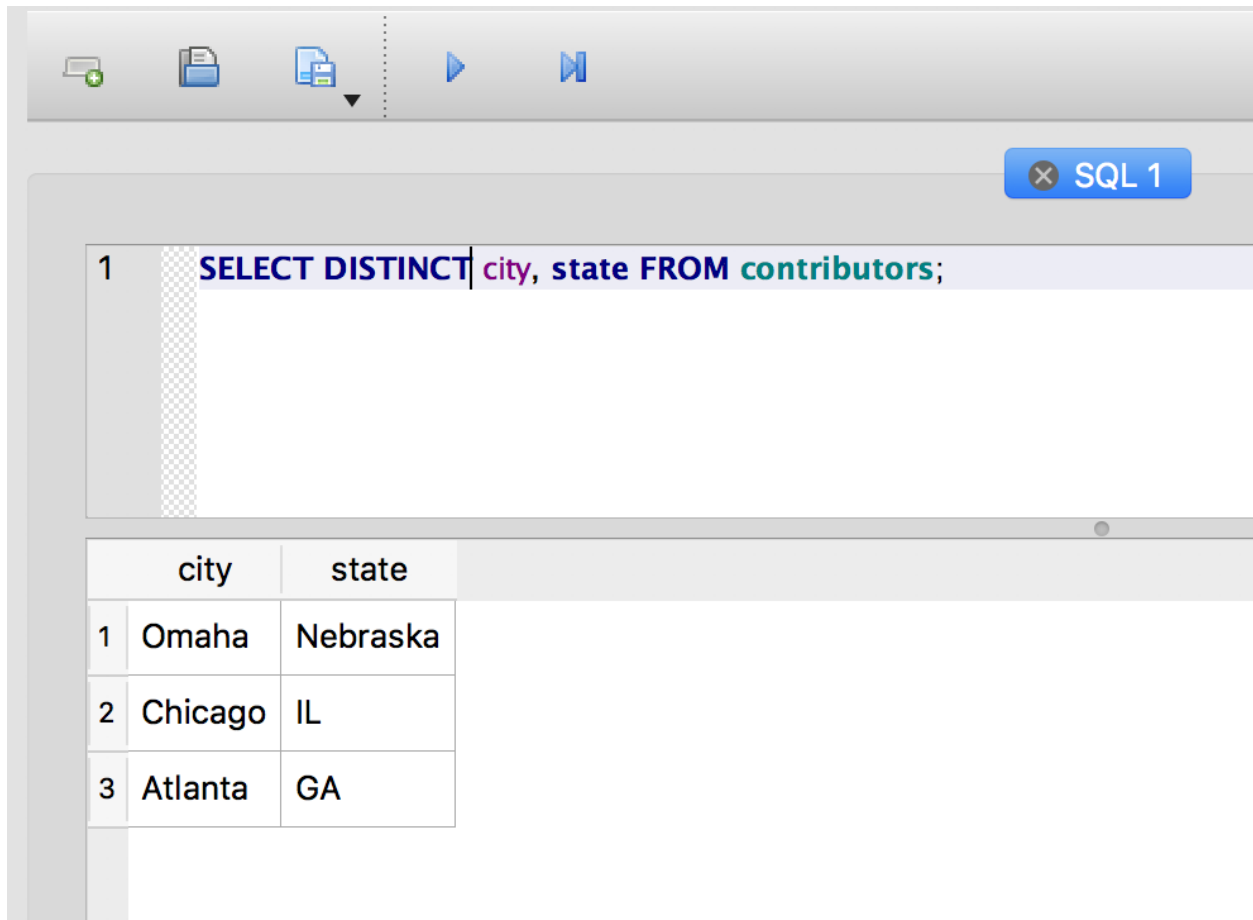
Below the query, the results are displayed in a table with two columns: "city" and "state". The results are as follows:

|   | city    | state    |
|---|---------|----------|
| 1 | Omaha   | Nebraska |
| 2 | Chicago | IL       |
| 3 | Atlanta | GA       |
| 4 | Atlanta | GA       |

## 1.7 Using DISTINCT to get a distinct set

The SELECT query above gives us a list of cities and states, but it includes duplicate rows for Atlanta, GA. Adding DISTINCT to the query eliminates the duplicates:

```
SELECT DISTINCT city, state FROM contributors;
```



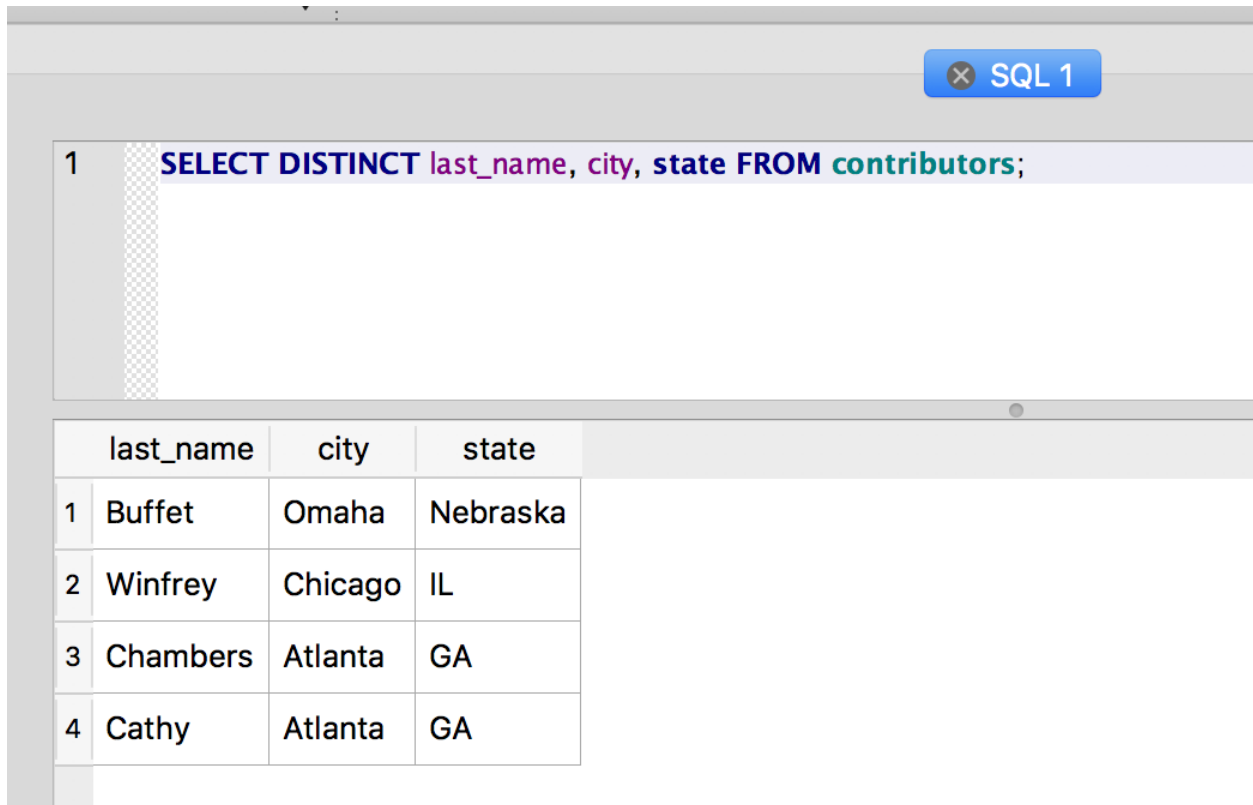
The screenshot shows a SQL IDE interface. At the top, there is a toolbar with icons for file operations and execution. Below the toolbar, a tab labeled 'SQL 1' is active. The main area displays a SQL query: `SELECT DISTINCT city, state FROM contributors;`. Below the query, the results are shown in a table with two columns: 'city' and 'state'. The table contains three rows of data.

|   | city    | state    |
|---|---------|----------|
| 1 | Omaha   | Nebraska |
| 2 | Chicago | IL       |
| 3 | Atlanta | GA       |

Now you should have only three rows in your results, showing the unique combinations for city and state in the table.

Notice what happens if you add the `last_name` field to the `DISTINCT` query:

```
SELECT DISTINCT last_name, city, state FROM contributors;
```



The screenshot shows a SQL editor window with a tab labeled 'SQL 1'. The query entered is: `1 SELECT DISTINCT last_name, city, state FROM contributors;`. Below the query, the results are displayed in a table with four columns: an index, `last_name`, `city`, and `state`. The results table contains four rows of data.

|   | last_name | city    | state    |
|---|-----------|---------|----------|
| 1 | Buffet    | Omaha   | Nebraska |
| 2 | Winfrey   | Chicago | IL       |
| 3 | Chambers  | Atlanta | GA       |
| 4 | Cathy     | Atlanta | GA       |

We’re back to four rows. There are four distinct combinations of `last_name`, `city` and `state` in the table, so that’s what we get from `DISTINCT`.

## 1.8 The WHERE Clause

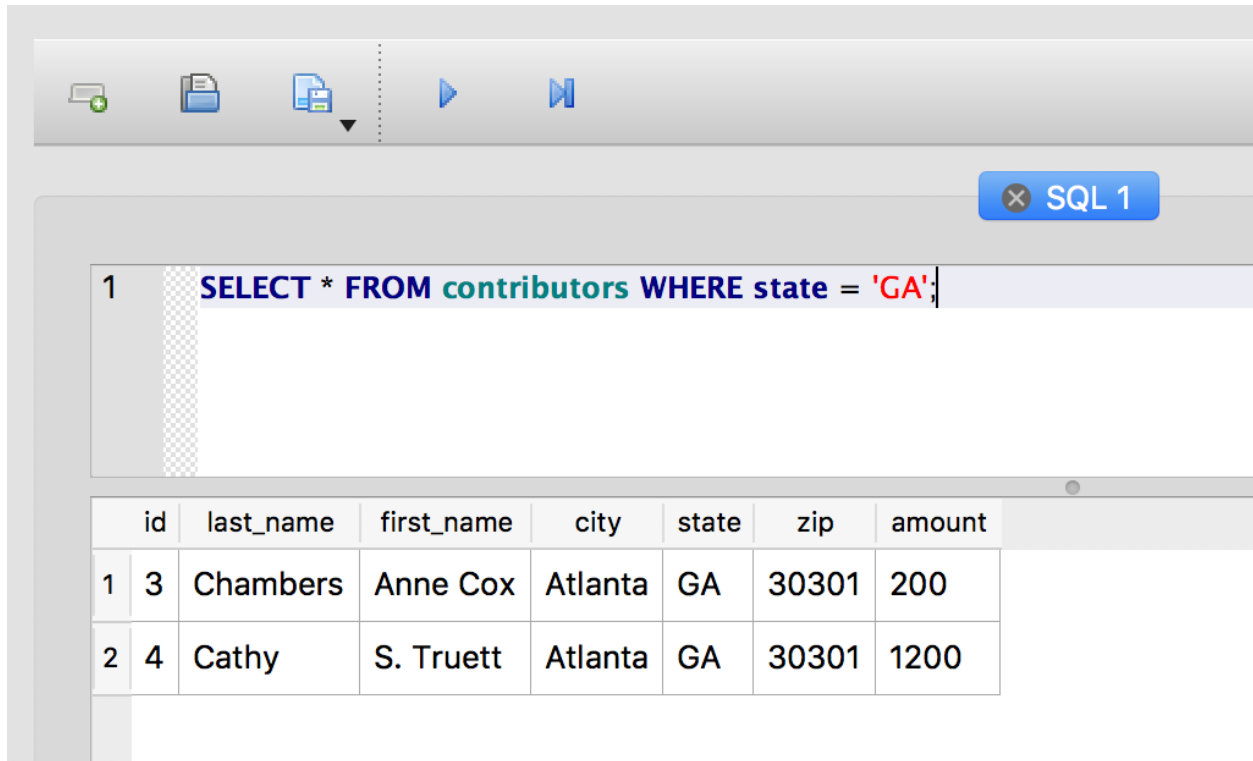
The `WHERE` clause provides the scalpel for your SQL operations. A well-crafted `WHERE` clause can let you take exactly the slice of the data you want. It sets the conditions for the `SELECT`, and the query will return only those rows that match the conditions.

Say, for example, we only wanted to see contributors from Georgia:

```
SELECT * FROM contributors WHERE state='GA';
```

Remember the single quotes around the string “GA”





And you can test for more than equality in the `WHERE` clause. This query finds only the contributors who have donated more than \$1200:

```
SELECT * FROM contributors WHERE amount > 1200;
```

Of course, donors who have given exactly \$1200 won't be included in the results. To include them, use the `>=` operator:

```
SELECT * FROM contributors WHERE amount >= 1200;
```

Here are some other operators you can use:

| operator | description           |
|----------|-----------------------|
| =        | Equal                 |
| !=       | Not equal*            |
| >        | Greater than          |
| <        | Less than             |
| >=       | Greater than or equal |
| <=       | Less than or equal    |

\* Many database systems also use `<>` for “Not equal”

## 1.9 Combining conditions with AND and OR

You can combine conditions using `AND` and `OR`. For example, let's find all contributors from Georgia who have given more than \$1000:

```
SELECT * FROM contributors WHERE state = 'GA' AND amount > 1000;
```

SQL 1

1 `SELECT * FROM contributors WHERE state = 'GA' AND amount > 1000;`

|   | id | last_name | first_name | city    | state | zip   | amount |
|---|----|-----------|------------|---------|-------|-------|--------|
| 1 | 4  | Cathy     | S. Truett  | Atlanta | GA    | 30301 | 1200   |

Now let's find all contributors who either live in Georgia or who have given more than \$1000:

```
SELECT * FROM contributors WHERE state = 'GA' OR amount > 1000;
```

SQL 1

1 `SELECT * FROM contributors WHERE state = 'GA' OR amount > 1000;`

|   | id | last_name | first_name | city    | state    | zip   | amount |
|---|----|-----------|------------|---------|----------|-------|--------|
| 1 | 1  | Buffet    | Warren     | Omaha   | Nebraska | 68101 | 1500   |
| 2 | 3  | Chambers  | Anne Cox   | Atlanta | GA       | 30301 | 200    |
| 3 | 4  | Cathy     | S. Truett  | Atlanta | GA       | 30301 | 1200   |

And now let's try to get the big spenders from Chicago and Georgia:

```
SELECT * FROM contributors WHERE city = 'Chicago' OR state = 'GA' AND amount > 1000;
```

SQL 1

1 SELECT \* FROM contributors WHERE city = 'Chicago' OR state = 'GA' AND amount > 1000;

|   | id | last_name | first_name | city    | state | zip   | amount |
|---|----|-----------|------------|---------|-------|-------|--------|
| 1 | 2  | Winfrey   | Oprah      | Chicago | IL    | 60601 | 500    |
| 2 | 4  | Cathy     | S. Truett  | Atlanta | GA    | 30301 | 1200   |

Hmm ... Oprah is in the list, but she only donated \$500. What gives?

The problem here is that the `AND` operator has a higher precedence than the `OR` operator, which means it gets evaluated first. So, in effect, our query really looks like this:

```
SELECT * FROM contributors WHERE city = 'Chicago' OR (state = 'GA' AND amount > 1000);
```

Which selects all contributors from Chicago and only those contributors from Georgia who have also donated more than \$1000.

We can use parentheses to clarify the original query and actually get the high rollers we wanted:

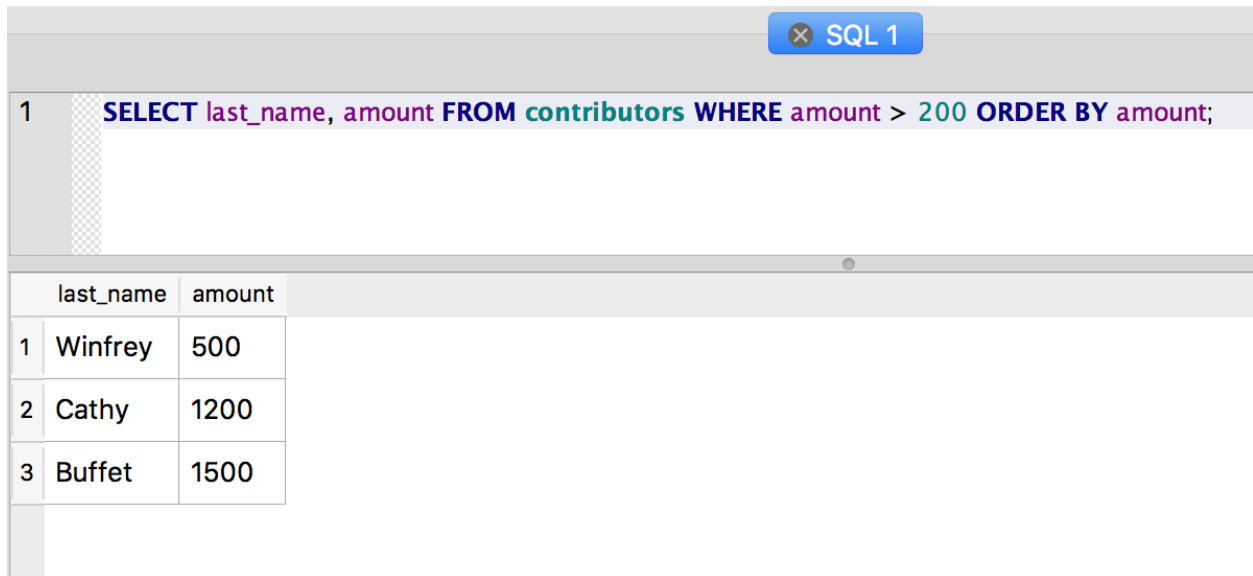
```
SELECT * FROM contributors WHERE (city = 'Chicago' OR state = 'GA') AND amount > 1000;
```

Parentheses are often helpful when you need to disambiguate a query. Technically, you're changing the order of evaluation here, but you're also just making the intention of your statement clear.

## 1.10 Sorting results with `ORDER BY`

To order your result set by the values in a particular column, use `ORDER BY`:

```
SELECT last_name, amount FROM contributors WHERE amount > 200 ORDER BY amount;
```



SQL 1

```
1 SELECT last_name, amount FROM contributors WHERE amount > 200 ORDER BY amount;
```

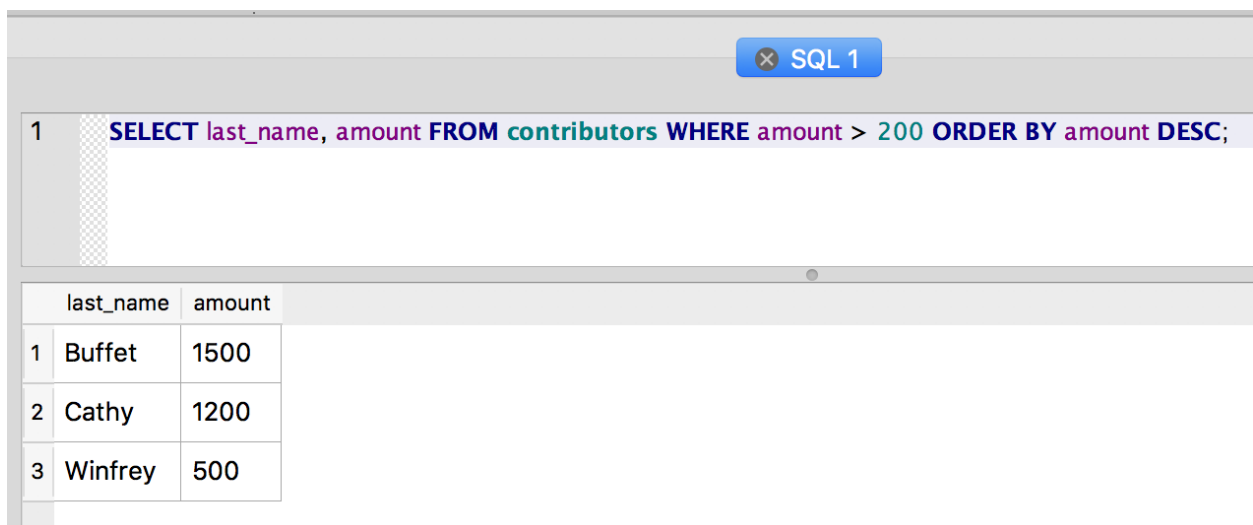
|   | last_name | amount |
|---|-----------|--------|
| 1 | Winfrey   | 500    |
| 2 | Cathy     | 1200   |
| 3 | Buffet    | 1500   |

Only the rows matching the WHERE clause are returned (i.e. only those with an amount exceeding \$200).

The default direction for ORDER BY is ascending; results are ordered from smallest amount to greatest.

To specify the direction of the sorting, use the DESC or ASC keyword:

```
SELECT last_name, amount FROM contributors WHERE amount > 200 ORDER BY amount DESC;
```



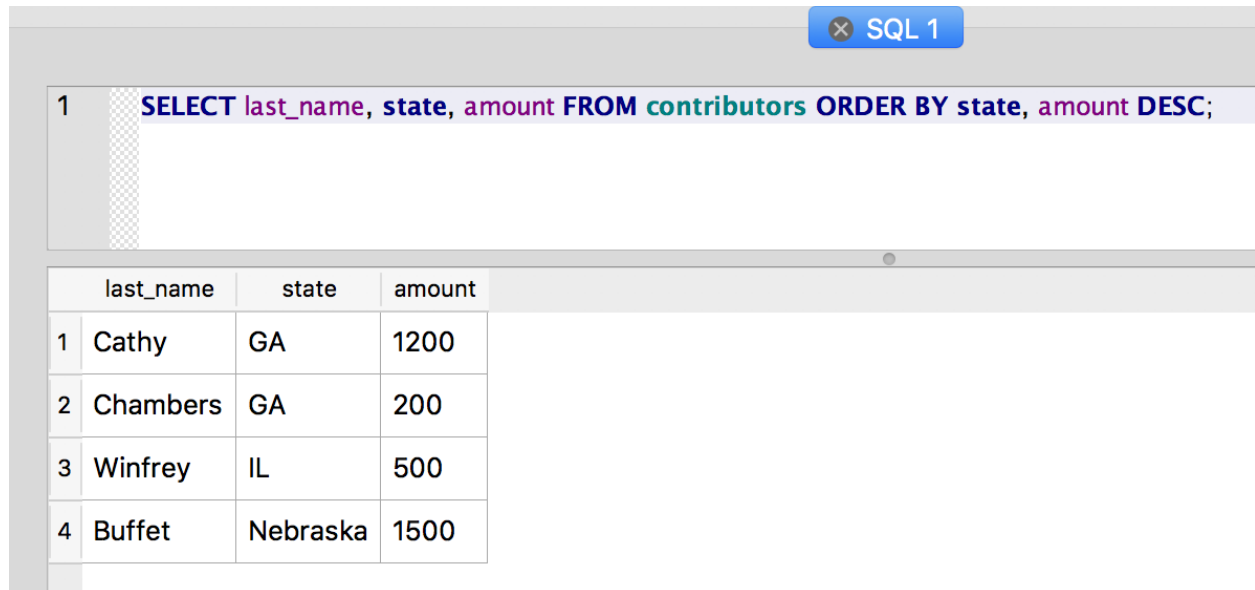
SQL 1

```
1 SELECT last_name, amount FROM contributors WHERE amount > 200 ORDER BY amount DESC;
```

|   | last_name | amount |
|---|-----------|--------|
| 1 | Buffet    | 1500   |
| 2 | Cathy     | 1200   |
| 3 | Winfrey   | 500    |

You can also order the results by more than one column. Rows with the same value for the first column of the ORDER BY are further ordered by the additional column(s):

```
SELECT last_name, state, amount FROM contributors ORDER BY state, amount DESC;
```



The screenshot shows a SQL editor window with a tab labeled 'SQL 1'. The query entered is: `SELECT last_name, state, amount FROM contributors ORDER BY state, amount DESC;`. Below the query, the results are displayed in a table with four columns: an index, last\_name, state, and amount.

|   | last_name | state    | amount |
|---|-----------|----------|--------|
| 1 | Cathy     | GA       | 1200   |
| 2 | Chambers  | GA       | 200    |
| 3 | Winfrey   | IL       | 500    |
| 4 | Buffet    | Nebraska | 1500   |

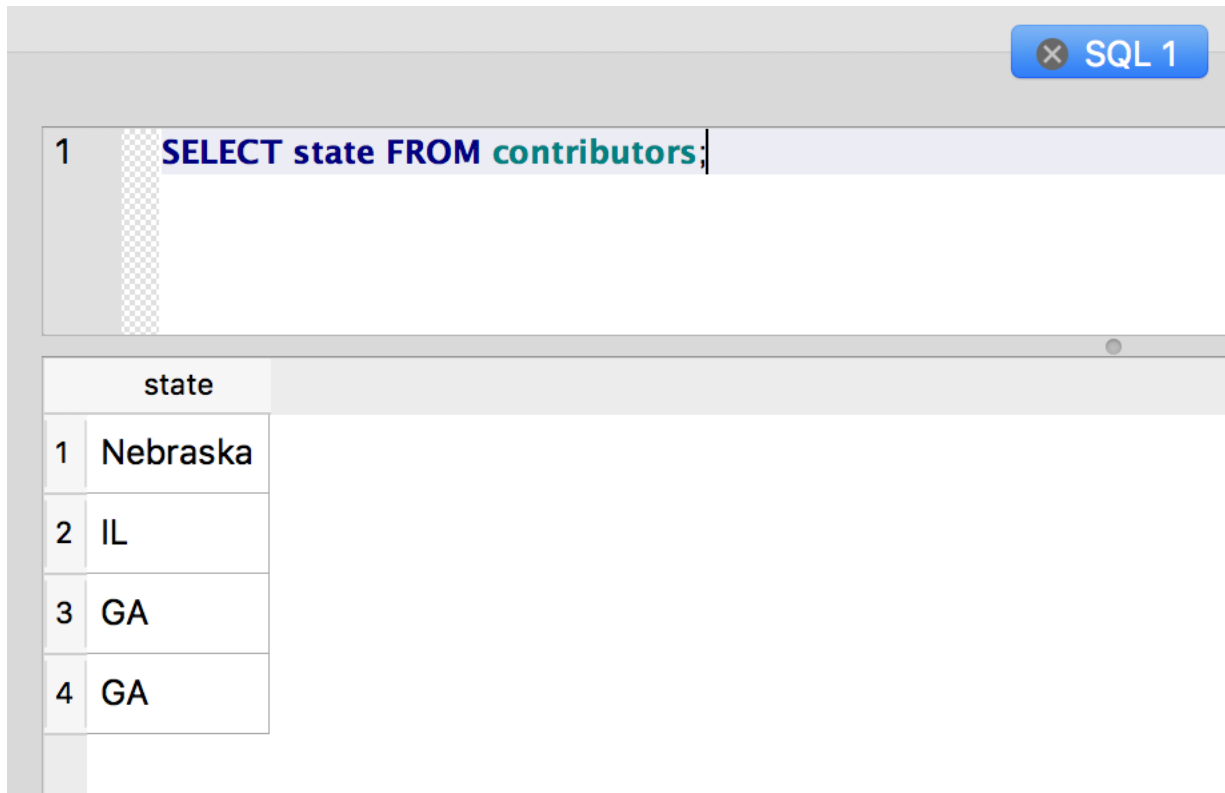
Here we get the list of contributors ordered by state and then ordered by the amount, from highest to lowest amount within the state, of their contribution. This is one quick way to see who has contributed the most in each state.

## 1.11 Changing values with UPDATE

Now we have some basic skills for creating tables, inserting data into the table and querying the data we've inserted. But what about changing the values in existing rows? To change the value of existing rows, we use the `UPDATE` statement.

One thing that just looks wrong with our data set is that value "Nebraska" in the state column:

```
SELECT state FROM contributors;
```



The screenshot shows a SQL editor window with a title bar that says "SQL 1". Inside the editor, the first line of a query is visible: `1 SELECT state FROM contributors;`. Below the editor, a table displays the results of the query. The table has a single column labeled "state" and contains four rows of data.

|   | state    |
|---|----------|
| 1 | Nebraska |
| 2 | IL       |
| 3 | GA       |
| 4 | GA       |

That should be the postal abbreviation, like the other rows. To change that value, we need to use `UPDATE` to set a new value for the column. **But we want to make sure we don't blow away the state values in our other columns.**

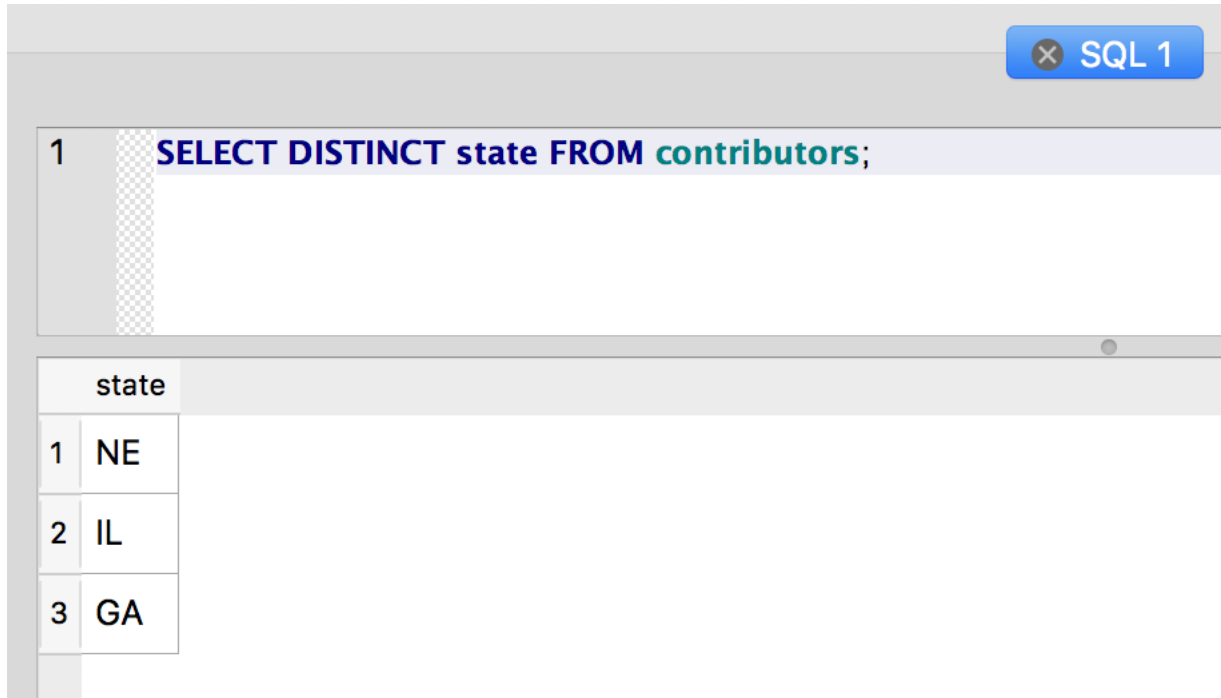
If we just used `UPDATE contributors SET state = 'NE';` - **DON'T EXECUTE THIS!!** - we would end up replacing the state value in every row with "NE". Not exactly what we want.

So, we have to define a `WHERE` clause to determine which rows will be changed by the `UPDATE`:

```
UPDATE contributors SET state = 'NE' WHERE state = 'Nebraska';
```

Ok, let's see how the state list looks:

```
SELECT DISTINCT state FROM contributors;
```



Now that's more like it.

## 1.12 Deleting data

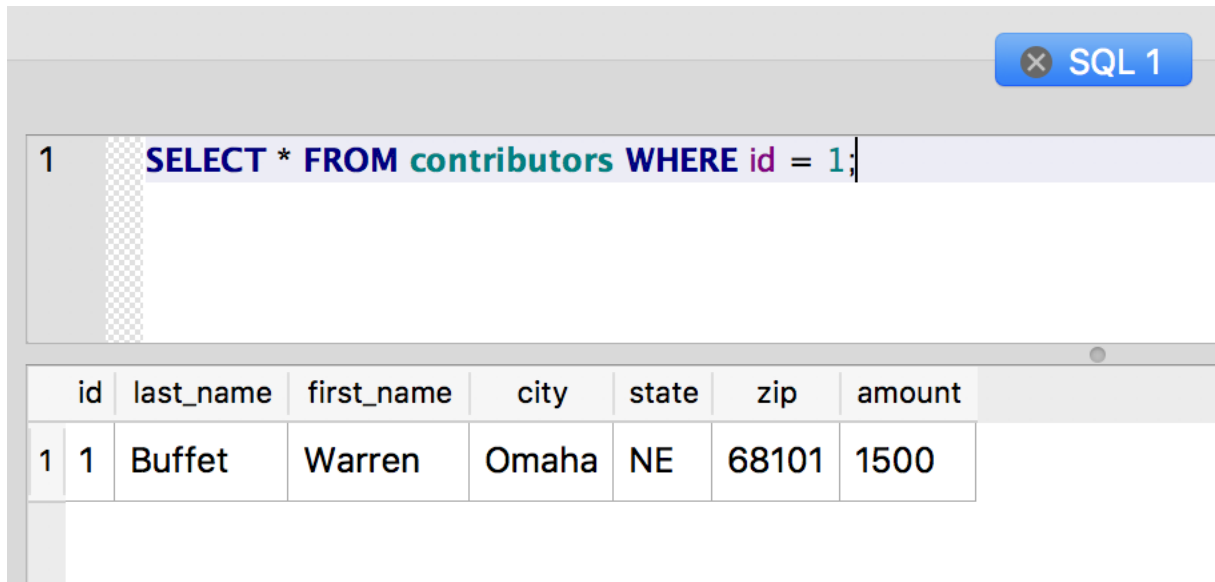
The remaining keyword from the original list is `DELETE`, which unsurprisingly deletes rows from the table. As when using `UPDATE`, it's important to specify a `WHERE` clause with `DELETE`. Running `DELETE` without a `WHERE` clause will blow away your precious data and can seriously ruin your day.

Before executing a `DELETE` or `UPDATE`, it's always a good idea to run a `SELECT` with the same `WHERE` clause, just to see which rows your changes will affect.

So, let's get rid of one of our rows. How about deleting Warren Buffet?

For our `WHERE` clause, we could match on any column or combination of columns, but if we know the `PRIMARY KEY` value of the row, that's our safest bet. Because it's a unique identifier, we can be certain we're not accidentally deleting other rows. First let's make sure we have the row we want:

```
SELECT * FROM contributors WHERE id = 1;
```



The screenshot shows a SQL editor window with a tab labeled "SQL 1". The query entered is:

```
1 SELECT * FROM contributors WHERE id = 1;
```

Below the query, the results are displayed in a table:

|   | id | last_name | first_name | city  | state | zip   | amount |
|---|----|-----------|------------|-------|-------|-------|--------|
| 1 | 1  | Buffet    | Warren     | Omaha | NE    | 68101 | 1500   |

Looks like the one we want, so let's delete it:

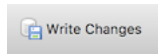
```
DELETE FROM contributors WHERE id = 1;
```

Notice that we don't need to specify columns or use \* with DELETE, since we're deleting the entire row.

Now the row should be gone:

```
SELECT * FROM contributors;
```

Finally, you should save the database changes you've made so you don't lose your work. You can save the changes by clicking the "Write Changes" button:



## 1.13 Further References

[http://www.w3schools.com/sql/sql\\_intro.asp](http://www.w3schools.com/sql/sql_intro.asp)

<http://www.firstsql.com/tutor.htm>

<https://hackr.io/tutorials/learn-sql>



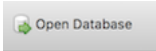
*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](https://creativecommons.org/licenses/by-nc-nd/3.0/).



Now that we have some dirty data and a few keywords, we can start to write some more interesting queries. In the process, we'll learn a few of the idiosyncrasies of SQL.

## 2.1 Getting started

For this section, we'll use a pre-packaged database containing 103 rows of FEC data, with some light modifications for the purposes of learning.

- Download `contributors.db`
- Fire up *DB Browser* if you don't already have it open
- Click “Open Database”: 
- Navigate to the “contributors.db” and open it

Now you're ready to dive deeper into SQL's features for querying and manipulating data, starting with how to handle *null values*.

## 2.2 Nothing can come of nothing: Using IS NULL

Let's take a look back at our original CREATE statement for the contributors table:

```
CREATE TABLE `contributors` (  
  `id`      INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
  `last_name` TEXT,  
  `first_name` TEXT,  
  `city`    TEXT,  
  `state`   TEXT,  
  `zip`     TEXT,
```

(continues on next page)

(continued from previous page)

```
`amount`      INTEGER
);
```

Notice that we defined the `id` column as `NOT NULL`, which meant that it was a required field. Because that field is serving as our unique identifier or **PRIMARY KEY** for the row, it can't be empty.

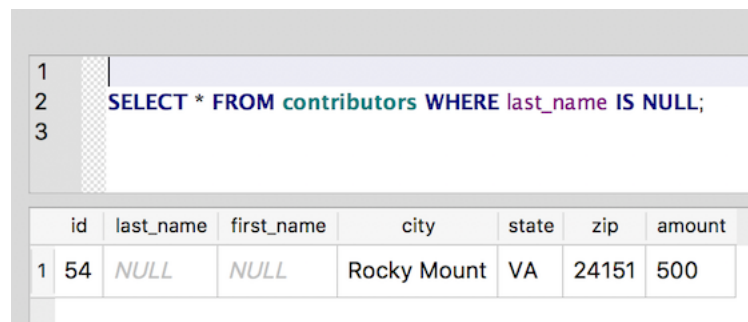
The keyword `NULL` is a special value in SQL. It's a placeholder for an empty field. If a field is `NULL`, it's really empty. That means it's not 0. It's not an empty string (""). If you're of a philosophical mind, you might call `NULL` the “nothing that is”. If you're of a pragmatic mind, you might just think of it as a placeholder where no value has been entered.

But being nothing (or a placeholder for an empty value) comes with a cost. `NULL` can't be compared with other data types such as strings. And we can't use normal operators to match it, either. So `=`, `!=` and friends don't work with `NULL`. Don't believe me? Try it out:

```
SELECT * FROM contributors WHERE last_name = NULL;
```

Instead, to query for null values, we use the keywords `IS NULL`:

```
SELECT * FROM contributors WHERE last_name IS NULL;
```



The screenshot shows a SQL query editor with a light blue background. The query `SELECT * FROM contributors WHERE last_name IS NULL;` is entered on line 2. Below the editor, a table of results is displayed with the following columns: `id`, `last_name`, `first_name`, `city`, `state`, `zip`, and `amount`. The table contains one row of data where `last_name` is `NULL`.

|   | id | last_name | first_name | city        | state | zip   | amount |
|---|----|-----------|------------|-------------|-------|-------|--------|
| 1 | 54 | NULL      | NULL       | Rocky Mount | VA    | 24151 | 500    |

`NULL`'s refusal to respond to normal operators can lead to some unforeseen effects. Take a look at this query, and guess what it should return:

```
SELECT * FROM contributors WHERE state = 'VA' AND last_name != 'Lewis';
```

(Remember that `!=` means “is not equal.”)

There are three contributors from VA in the table:

- Robert Albrecht,
- Donald S. Lewis
- and someone from Rocky Mount whose name fields are empty. (Yes, the data did come in like this from the FEC.)

You can see the list by using “Browse Data” and filtering the `state` field for “VA”, or by running this query:

```
SELECT * FROM contributors WHERE state = 'VA';
```

So, the clause `WHERE state = 'VA' AND last_name != 'Lewis'` looks like it's asking for all contributors from Virginia whose last name is not Lewis. And it looks like it should return both Albrecht and the Rocky Mount contributor. But when we run it (cue “Price Is Right” sad horn sound), we only get Albrecht:

```

1 SELECT * FROM contributors WHERE state = 'VA' AND last_name != 'Lewis';
2

```

|   | id | last_name | first_name | city       | state | zip   | amount |
|---|----|-----------|------------|------------|-------|-------|--------|
| 1 | 8  | Albrecht  | Robert     | woodbridge | VA    | 22192 | 10     |

“Curiouser and curiouser,” you might say. This makes strict logical sense when we consider that the NULL data type can’t be compared with any other data type, but really it does seem a bit of a pain (even to some of the SQL gurus). The solution is to use `IS NULL`. Here’s one way to write the query to get the results we intended:

```
SELECT * FROM contributors WHERE state = 'VA' AND (last_name != 'Lewis' OR last_name_
↪ IS NULL);
```

(The parentheses are optional here, but they do help express our intentions.)

And now we get the two expected result rows:

```

1 SELECT * FROM contributors
2 WHERE state = 'VA' AND (last_name != 'Lewis' OR last_name IS NULL);
3

```

|   | id | last_name | first_name | city        | state | zip   | amount |
|---|----|-----------|------------|-------------|-------|-------|--------|
| 1 | 8  | Albrecht  | Robert     | woodbridge  | VA    | 22192 | 10     |
| 2 | 54 | NULL      | NULL       | Rocky Mount | VA    | 24151 | 500    |

## 2.2.1 IS NOT NULL

The opposite of `IS NULL` is (drumroll) ... `IS NOT NULL`. And it works pretty much as we’d expect:

```
SELECT * FROM contributors WHERE state = 'VA' AND last_name IS NOT NULL;
```

```

1 SELECT * FROM contributors WHERE state = 'VA' AND last_name IS NOT NULL;
2
3

```

|   | id  | last_name | first_name | city           | state | zip       | amount |
|---|-----|-----------|------------|----------------|-------|-----------|--------|
| 1 | 8   | Albrecht  | Robert     | woodbridge     | VA    | 22192     | 10     |
| 2 | 102 | Lewis     | Donald S.  | Virginia Beach | VA    | 234513838 | 250    |

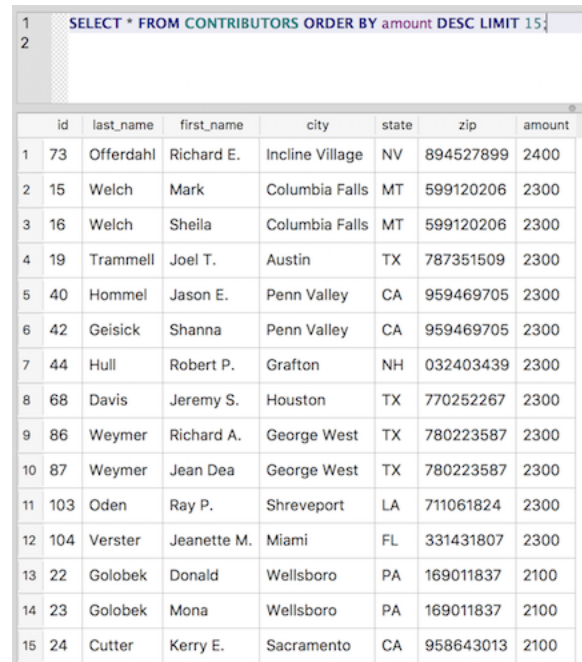
This negative form is pretty handy for filtering null values from the results set.

## 2.3 Knowing your limitations: Using LIMIT

So far, all of our queries have returned the full result set of rows matching the `WHERE` clause. But sometimes you only want a subset of the results. Let’s use the `LIMIT` keyword to get the top 15 contributors by contribution.

First we order the results by amount (in descending order), and then we limit the results to only the first 15 rows:

```
SELECT * FROM CONTRIBUTORS ORDER BY amount DESC LIMIT 15;
```



The screenshot shows a SQL query editor with the query `SELECT * FROM CONTRIBUTORS ORDER BY amount DESC LIMIT 15;` entered. Below the editor, a table displays the results of the query, ordered by amount in descending order. The table has 7 columns: `id`, `last_name`, `first_name`, `city`, `state`, `zip`, and `amount`. The results are as follows:

|    | id  | last_name | first_name  | city            | state | zip       | amount |
|----|-----|-----------|-------------|-----------------|-------|-----------|--------|
| 1  | 73  | Offerdahl | Richard E.  | Incline Village | NV    | 894527899 | 2400   |
| 2  | 15  | Welch     | Mark        | Columbia Falls  | MT    | 599120206 | 2300   |
| 3  | 16  | Welch     | Sheila      | Columbia Falls  | MT    | 599120206 | 2300   |
| 4  | 19  | Trammell  | Joel T.     | Austin          | TX    | 787351509 | 2300   |
| 5  | 40  | Hommel    | Jason E.    | Penn Valley     | CA    | 959469705 | 2300   |
| 6  | 42  | Geisick   | Shanna      | Penn Valley     | CA    | 959469705 | 2300   |
| 7  | 44  | Hull      | Robert P.   | Grafton         | NH    | 032403439 | 2300   |
| 8  | 68  | Davis     | Jeremy S.   | Houston         | TX    | 770252267 | 2300   |
| 9  | 86  | Weymer    | Richard A.  | George West     | TX    | 780223587 | 2300   |
| 10 | 87  | Weymer    | Jean Dea    | George West     | TX    | 780223587 | 2300   |
| 11 | 103 | Oden      | Ray P.      | Shreveport      | LA    | 711061824 | 2300   |
| 12 | 104 | Verster   | Jeanette M. | Miami           | FL    | 331431807 | 2300   |
| 13 | 22  | Golobek   | Donald      | Wellsboro       | PA    | 169011837 | 2100   |
| 14 | 23  | Golobek   | Mona        | Wellsboro       | PA    | 169011837 | 2100   |
| 15 | 24  | Cutter    | Kerry E.    | Sacramento      | CA    | 958643013 | 2100   |

And if there aren't enough matching rows to reach the specified limit, the limit is simply ignored:

```
SELECT * FROM contributors WHERE amount > 2100 LIMIT 15;
```

## 2.4 Casting a wider net with LIKE

While it's helpful to be able to write queries that look for equality (`last_name = 'Smith'`) or inequality (`last_name != 'Smith'`), sometimes you want to do something a little messier, such as looking for everyone whose last name starts with 'T'. Or maybe you want to look for matches to a five-digit ZIP code, but some of your rows use ZIP+4. For these kinds of expressions, you can use the `LIKE` operator, which will perform a partial match.

A brief aside worth mentioning: The `LIKE` operator is case-insensitive for English letters, so a query for "SMITH" or "smith" would both match the name "Smith."

To perform a partial match using `LIKE`, you can combine normal characters and special wildcard characters to construct a pattern. For example, the percent sign (`%`) will match any sequence of zero or more characters. So to match any zip that begins with 77566, we can use this statement:

```
SELECT zip FROM contributors WHERE zip LIKE '77566%';
```

```
1 SELECT zip FROM contributors WHERE zip LIKE '77566%';
```

| zip |           |
|-----|-----------|
| 1   | 775661497 |
| 2   | 775666036 |

Notice that it matches both 775661497 and 77566036. It would also match 77566, because the % will match zero characters, too.

The % is probably the most common special character used in pattern matching with LIKE. Another less commonly used pattern matcher is the underscore (“\_”), which matches any single character in the string.

Say, for example, we wanted to start cleaning our data, and we wanted to remove the middle initials from the first\_name field and put them into a new middle\_name column. (This sort of thing can get tricky very quickly, but for now we’ll trip along happily assuming everything goes smoothly.)

As a first step, we want simply to examine all of the rows that appear to contain middle initials in first\_name. Here’s a query that will get us at least part of the way there:

```
SELECT * FROM contributors WHERE first_name LIKE '% _.';
```

Reading patterns like this one may prove a little tricky at first, but in time ... who am I kidding, it’s still pretty tricky, but you can figure it out. Let’s break it down:

- The pattern starts with %, which we know means “match any series of zero or more characters,” which is pretty much anything.
- Next we have a space. It’s hard to see, but it’s between the % and the underscore ( \_ ). So we’re matching anything plus a space.
- Then we have the magic underscore ( \_ ), meaning any single character.
- And finally, we have a period ( . ), which is just a literal period here.

And here’s the result (you should get 60 rows, but we’ve truncated the results here):

```
1 SELECT * FROM contributors WHERE first_name LIKE '% _.';
```

|    | id | last_name | first_name | city         | state | zip       | amount |
|----|----|-----------|------------|--------------|-------|-----------|--------|
| 1  | 17 | See       | Alvin B.   | Southwick    | MA    | 010770444 | 500    |
| 2  | 18 | Taylor    | Richard D. | Shreveport   | LA    | 711013114 | 250    |
| 3  | 19 | Trammell  | Joel T.    | Austin       | TX    | 787351509 | 2300   |
| 4  | 20 | Whitfield | George R.  | Seoul   Kore | ZZ    | 11070     | 1600   |
| 5  | 24 | Cutter    | Kerry E.   | Sacramento   | CA    | 958643013 | 2100   |
| 6  | 25 | Miller    | James R.   | Burtonsville | MD    | 208661043 | 2100   |
| 7  | 28 | Howard    | Patrick G. | Concord      | NH    | 033022219 | 2100   |
| 8  | 29 | Shudlick  | Jon L.     | N Ft Myers   | FL    | 33917     | 500    |
| 9  | 30 | Furber    | Donna P.   | Downey       | CA    | 902402634 | 2100   |
| 10 | 31 | Furber    | James E.   | Downey       | CA    | 902402634 | 2100   |

So, in English, the pattern says to match “any series of characters followed by a space, a single character, and a period.”

This pattern will match things like:

- “John Q.”
- “1234 5.”
- “#\$%^ !.”
- ” B.”
- “J. B.”

It won’t, however, match the string “J. Quincy” because the period isn’t the last character in the field. Neither will it match “Alfred E. ” because we’ve left a space after the period.

To also match patterns that contain characters after the period, we would need to add a final % to the pattern:

```
SELECT * FROM contributors WHERE first_name LIKE '% _.%';
```

Now we’re matching the pattern “any series of zero or more characters, followed by a space, followed by a single character, followed by a period, followed by any series of zero or more characters.” (So, our little pattern expresses a pretty complex thought.)

Of course, we could just match any first\_name that contains a period, like this:

```
SELECT * FROM contributors WHERE first_name LIKE '%.%';
```

But then we also get names like “S. Truett,” which may or may not be what we intended.

Note: Some database systems include other wildcard characters to be used in patterns. For example, in some systems the pattern [xyz] will match one of the characters “x,” “y” or “z.” And the pattern [^xyz] will match any character that is *not* an “x,” “y” or “z. SQLite does not, by default, support this wildcard.

## 2.5 Using string functions: SUBSTR(), TRIM(), UPPER(), LOWER()

Using `LIKE` for partial matches can be pretty powerful, but as we’ve seen, patterns aren’t exactly beach reading. Another way to do partial matching is to use string functions to manipulate the values. String functions usually take the form of a keyword followed by parentheses. The parentheses contain any arguments we want to pass to the function. The general format looks like this: `KEYWORD (ARG1, ARG2, ARG3)`. Usually the first argument is the string we want to manipulate. Here are some commonly used string functions:

### 2.5.1 SUBSTR()

The `SUBSTR()` function takes the string we hand it in the parentheses and returns a part of the string that we define (ergo, substring).

*As we’ll see with other string functions, this string argument can be - and typically is - the name of a column in a table. This gives us the power to manipulate all the values for a given column (or perhaps a limited subset).*

To determine which part of the string to return, `SUBSTR()` accepts a few additional arguments beyond the field that we’re targeting:

- the starting point of the desired substring (counting characters from the left)
- the number of characters to grab from that starting point

The full function call takes this form: `SUBSTR (STRING, START_POINT, LENGTH)`. The third argument is optional. If we leave it off, `SUBSTR()` returns all characters from the given starting point to the end of the string.

An example is probably more helpful. So, here is the ZIP query from earlier, rewritten to use a substring match in the `WHERE` clause of the query:

```
SELECT zip FROM contributors WHERE SUBSTR(zip, 1, 5) = '77566';
```

Above, we’re asking for all ZIP codes in the table whose first five characters match ‘77566’. This query will return the same result set we saw earlier: 775661497 and 77566036.

Functions can also be used in the `SELECT` clause of the query, so we can do something like this:

```
SELECT SUBSTR(zip, 1, 5) FROM contributors;
```

Now we’re getting the five-digit representation of all ZIPs in the table (and dropping the extra four digits from the ZIP+4s):

```
1 SELECT SUBSTR(zip, 1, 5) FROM contributors;
```

| SUBSTR(zip, 1, 5) |       |
|-------------------|-------|
| 1                 | 60601 |
| 2                 | 30301 |
| 3                 | 30301 |
| 4                 | 20012 |
| 5                 | 07417 |
| 6                 | 02563 |
| 7                 | 22192 |
| 8                 | 07417 |

## 2.5.2 TRIM()

The `TRIM()` function is most frequently used to trim white space from either side of a string. During data entry, strings are often accidentally inserted with leading or trailing whitespace. To simulate this case, let's mess up the data even more:

```
UPDATE contributors SET state = ' GA ' WHERE last_name = 'Cathy';
```

Now try selecting all rows where the state field is equal to 'GA' (with no extra surrounding spaces around the state postal):

```
select * from contributors WHERE state = 'GA';
```

```
1 select * from contributors WHERE state = 'GA';
```

|   | id | last_name | first_name | city    | state | zip       | amount |
|---|----|-----------|------------|---------|-------|-----------|--------|
| 1 | 3  | Chambers  | Anne Cox   | Atlanta | GA    | 30301     | 200    |
| 2 | 48 | Dermer    | Jeffrey D. | Atlanta | GA    | 303097662 | 500    |

So, now Cathy isn't appearing in our list of Georgians. Even worse, we've created a new state:



```
SELECT DISTINCT state FROM contributors;
```

```
1 SELECT DISTINCT state FROM contributors;
```

|   | state |
|---|-------|
| 1 | IL    |
| 2 | GA    |
| 3 | GA    |
| 4 | DC    |

We can use `TRIM()` to clean things up:

```
UPDATE contributors SET state = TRIM(state);
```

Notice here that we’re not using a `WHERE` clause on the `UPDATE` statement. This means that all rows will be updated, which is usually not what you want at all. Consider if we had used `SET state = 'GA'` in the statement above; we’d now have a table full of Georgians and a mess to clean up. Because we’re using a function, rather than a literal string here, we can update everything at once, trimming the white space from the front and end of every state value. The function operates on the value in the state column for each row in turn.

And now we’re back to normal:

```
SELECT DISTINCT state FROM contributors;
```

```
1 SELECT DISTINCT state FROM contributors;
```

|   | state |
|---|-------|
| 1 | IL    |
| 2 | GA    |
| 3 | DC    |
| 4 | NJ    |
| 5 | MA    |
| 6 | VA    |

The `TRIM()` function can also be used to strip characters other than spaces from the front and end of a string, although this usage is probably less common. To tell `TRIM()` which characters to remove, pass a second argument which contains any characters to be removed. For example, `TRIM (state, '.,')` would remove any periods or commas appearing at the beginning or end of the state name (i.e. “GA.” would become “GA”).

### 2.5.3 UPPER() and LOWER()

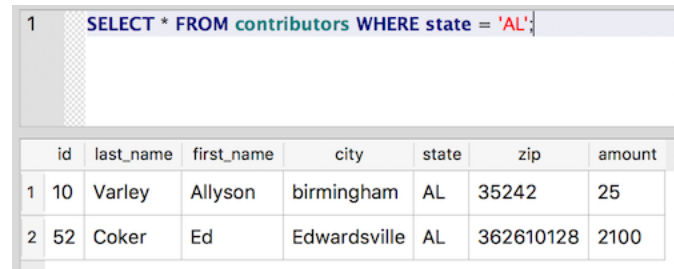
Another common problem in dirty data is inconsistencies in capitalization. For example, let's find all of the contributors from Birmingham, Alabama:

```
SELECT * FROM contributors WHERE state = 'AL' AND city = 'Birmingham';
```

Hmm ... apparently there aren't any.

But when we check on all contributors from Alabama, we get a different story:

```
SELECT * FROM contributors WHERE state = 'AL';
```



| 1 | SELECT * FROM contributors WHERE state = 'AL'; |           |            |              |       |           |        |
|---|--|-----------|------------|--------------|-------|-----------|--------|
|   | id   | last_name | first_name | city         | state | zip       | amount |
| 1 | 10   | Varley    | Allyson    | birmingham   | AL    | 35242     | 25     |
| 2 | 52   | Coker     | Ed         | Edwardsville | AL    | 362610128 | 2100   |

So, the problem is that Birmingham isn't properly capitalized. Now, we could do a SELECT using `city = 'birmingham'`, but then we'd miss any rows that properly capitalize the city name. And what about rows that use ALL CAPS?

An easy way to get around these issues of case-sensitivity is to use the `UPPER()` or `LOWER()` string functions to standardize capitalization on the values:

```
SELECT * FROM contributors WHERE UPPER(city) = 'BIRMINGHAM';
```

The `UPPER()` function translates each letter in the city value to upper case.

Note that we are *not* changing the values in this column to upper-case. Instead, we're dynamically modifying the values in our `WHERE` clause purely for the purposes of matching records in a select query, leaving the original values unchanged.

As a result, this query will give us the lower-case version, but it will also match "Birmingham" and "BIRMINGHAM" (not to mention "BIRMinGham"), as they will all be rendered as "BIRMINGHAM" by `UPPER()`.

Note: By default *LIKE* is *not* case-sensitive in SQLite, but that is not true of all database management systems. Also, in some other database systems, such as MySQL, the basic equality operator (`=`) is case insensitive, but that's not true in SQLite, and it isn't true in other systems. When in doubt, it's safer to use `LOWER()` or `UPPER()` to ensure case insensitivity. (Also, some databases use `UCASE()` and `LCASE()` rather than `UPPER()` and `LOWER()`.)

## 2.6 Pull yourself together: The concatenate operator (||)

Sometimes we want to combine values from different columns, either in the `WHERE` clause or for the results. SQLite uses double-pipes (`||`) - more formally known as the concatenation operator - to combine strings. You can combine both literal strings (in quotation marks) and column values using this operator.

Say, for instance, we want a nicely formatted list of cities and states for contributors. To create a single result column that contains the city and state separated by a comma, we can use this query:

```
SELECT city || ', ' || state FROM contributors ORDER BY state, city;
```

We insert the comma and space as a literal string concatenated with the values from the city and state columns.

```
1 SELECT city || ', ' || state FROM contributors ORDER BY state, city;
```

|   | city    ', '    state |
|---|-----------------------|
| 1 | Edwardsville,AL       |
| 2 | birmingham,AL         |
| 3 | Bryant,AR             |
| 4 | Mesa,AZ               |
| 5 | Tubac,AZ              |
| 6 | Beverly Hills,CA      |

Note: Some other database management systems, such as MySQL use the `CONCAT()` function to perform concatenation. For example, `SELECT CONCAT (city, ', ', state) FROM contributors;` will *not* work in SQLite.

## 2.7 Pick One: Using BETWEEN and IN (NOT IN)

Often you'll want to get a value from within a range. The `BETWEEN` operator can do exactly that. Let's see which of our contributors has given between 500 and 1000 dollars:

```
SELECT * FROM contributors WHERE amount BETWEEN 500 AND 1000;
```

```
1 SELECT * FROM contributors WHERE amount BETWEEN 500 AND 1000;
```

|   | id | last_name   | first_name | city       | state | zip       | amount |
|---|----|-------------|------------|------------|-------|-----------|--------|
| 1 | 2  | Winfrey     | Oprah      | Chicago    | IL    | 60601     | 500    |
| 2 | 17 | See         | Alvin B.   | Southwick  | MA    | 010770444 | 500    |
| 3 | 21 | Young       | Craig      | Kyle       | TX    | 786400099 | 500    |
| 4 | 29 | Shudlick    | Jon L.     | N Ft Myers | FL    | 33917     | 500    |
| 5 | 32 | Baumgardner | David      | Wellington | TX    | 790954836 | 1000   |

Note: This query returns the same results as `SELECT * FROM contributors WHERE amount >= 500 AND amount <= 1000;` — but it's much more readable.

At other times, you may need to match values from within a set of choices. This is where the `IN` operator comes in handy. Let's find all contributors from a few southern states:

```
SELECT * FROM contributors WHERE state IN ('AL', 'GA', 'FL');
```

The choices are surrounded by parentheses and separated by commas. And don't forget the quote marks around literal strings. here's the result:

```
1 SELECT * FROM contributors WHERE state IN ('AL', 'GA', 'FL');
```

|   | id | last_name | first_name | city       | state | zip       | amount |
|---|----|-----------|------------|------------|-------|-----------|--------|
| 1 | 3  | Chambers  | Anne Cox   | Atlanta    | GA    | 30301     | 200    |
| 2 | 4  | Cathy     | S. Truett  | Atlanta    | GA    | 30301     | 1200   |
| 3 | 10 | Varley    | Allyson    | birmingham | AL    | 35242     | 25     |
| 4 | 29 | Shudlick  | Jon L.     | N Ft Myers | FL    | 33917     | 500    |
| 5 | 43 | Teasley   | Harry E.   | Tampa      | FL    | 336112814 | 1000   |

Again, you could have used a compound statement with `state = 'AL' OR state = 'GA' OR state = 'FL'` to achieve the same result, but the `IN` syntax makes things much clearer, and it's easier to write.

You can also use `NOT IN` to find results where a value is not included in the given set:

```
SELECT * FROM contributors WHERE state NOT IN ('CA', 'OR', 'AZ');
```

But beware that `NOT IN` won't work with null fields. So, if one of the rows has a null value for state, it would not be returned by the query above.

## 2.8 Aggregate Functions: COUNT, MAX, MIN, SUM, AVG

Aggregate functions allow us to perform calculations on values across rows. Using them, we can start to do some pretty interesting data analysis. To specify a column to use for the aggregate, pass the column name as the argument in parentheses: e.g. `COUNT (counted_column)`. Here's a quick run through some useful aggregate functions:

### 2.8.1 COUNT()

How many contributors do we have from California?

```
SELECT COUNT(id) FROM contributors WHERE state = 'CA';
```

| 1         | SELECT COUNT(id) FROM contributors WHERE state = 'CA'; |
|-----------|--|
| COUNT(id) |  |
| 1         | 17   |

The `COUNT(id)` function counts the number of unique ids. We could also have used `COUNT(*)`, which will count the number of rows. The result will be the same.

`COUNT()` can also be used with `DISTINCT` to return the number of distinct instances. For example, how many distinct ZIP Codes are there in the table?

```
SELECT COUNT(DISTINCT zip) FROM contributors;
```

Note that the `DISTINCT` keyword comes inside the parentheses. It is part of the argument passed to `COUNT()`.

```
1 SELECT COUNT(DISTINCT zip) FROM contributors;
```

|   | COUNT(DISTINCT zip) |
|---|---------------------|
| 1 | 92                  |

## 2.8.2 MIN() and MAX()

What is the maximum amount that any of our contributors has given?

```
SELECT MAX(amount) FROM contributors;
```

```
1 SELECT MAX(amount) FROM contributors;
```

|   | MAX(amount) |
|---|-------------|
| 1 | 2400        |

## 2.8.3 SUM()

What is the total amount of contributions from Georgia?

```
SELECT SUM(amount) FROM contributors WHERE state = 'GA';
```

```
1 SELECT SUM(amount) FROM contributors WHERE state = 'GA';
```

|   | SUM(amount) |
|---|-------------|
| 1 | 1900        |

## 2.8.4 AVG()

What is the average amount contributed?

```
SELECT AVG(amount) FROM contributors;
```

|             |                                       |
|-------------|---------------------------------------|
| 1           | SELECT AVG(amount) FROM contributors; |
| AVG(amount) |                                       |
| 1           | 1037.52427184466                      |

Of course, the usual caveats about using averages apply. I heard a nice example recently: “Which major at UNC produces graduates with the highest average salary?” Apparently, it was Geography - Michael Jordan’s major. Even if it isn’t true, it’s a nice warning about the way outliers can skew averages.

## 2.9 Beyond functions: Custom calculations

We’ve learned about many of the built-in functions that SQL provides for manipulating *strings* and summarizing data with *aggregates*. But what if SQL doesn’t provide a ready-made function for the task at hand?

Fortunately, SQL supports the ability to perform ad hoc calculations in the `SELECT` clause<sup>1</sup>.

Here’s a really basic example:

```
select 1 + 2
```

|       |              |
|-------|--------------|
| 1     | select 1 + 2 |
| 2     |              |
| 1 + 2 |              |
| 1     | 3            |

Simple, though not very useful. Things get more interesting when you start performing calculations on data.

Say that we wanted to know the average contribution amount for the entire data set.

We’ll pretend for now that we don’t already know about the built-in *AVG* function.

You can find that number by summing up all contributions and dividing by the total number of contributions:

```
SELECT sum(amount) * 1.0 / 103
FROM contributors;
```

---

<sup>1</sup> Custom calculations work in other clauses as well, such as in `WHERE` clauses. We focus on `SELECT` here because it’s one of the most common use cases for custom calculations.

```
1 select sum(amount) * 1.0 / 103
2 from contributors;
3
```

sum(amount) \* 1.0 / 103

```
1 1037.52427184466
```

Note that above, we've multiplied the sum of contributions ( `sum(amount)` ) by `1.0`. This forces the amount – which is an integer – to be treated as a decimal.

Failing to do so will result in SQLite dropping the numbers after the decimal, causing you to lose precision that may be important in a given query:

```
1 SELECT sum(amount) / 103
2 FROM contributors;
```

sum(amount) \* 1.0 / 103

```
1 1037.52427184466
```

Also note that we can make this query more flexible by updating the calculation to use the built-in `COUNT` function, instead of hard-coding the value.

```
SELECT sum(amount) * 1.0 / count(*)
FROM contributors;
```

```
1 select sum(amount) * 1.0 / count(*)
2 from contributors;
3
```

sum(amount) \* 1.0 / count(\*)

```
1 1037.52427184466
```

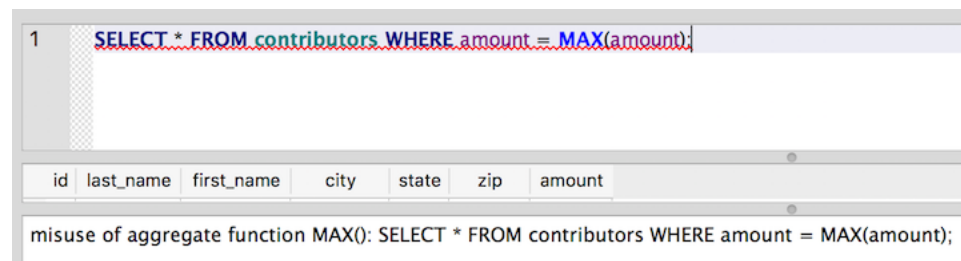
These examples are fairly simple, but hopefully they demonstrate SQL's flexibility. Built-in functions are quite handy, but there will likely come a time when you need a custom calculation - perhaps in combination with a built-in function - to get the job done.

## 2.10 Subqueries, the Russian dolls of SQL

When doing analysis, we often want to base one query on the results of another query. For example, we used the `MAX()` function to determine the maximum amount contributed. But what if we want to know who actually gave that maximum amount? We could try something like this:

```
SELECT * FROM contributors WHERE amount = MAX(amount);
```

But we won't like the results:



The screenshot shows a SQL query editor with a single line of code: `SELECT * FROM contributors WHERE amount = MAX(amount);`. Below the query, a table with columns `id`, `last_name`, `first_name`, `city`, `state`, `zip`, and `amount` is visible. At the bottom, a red error message reads: "misuse of aggregate function MAX(): SELECT \* FROM contributors WHERE amount = MAX(amount);".

We could also simply run two different queries, one to get the maximum amount, and another to find rows matching that amount:

```
SELECT MAX(amount) FROM contributors;
```

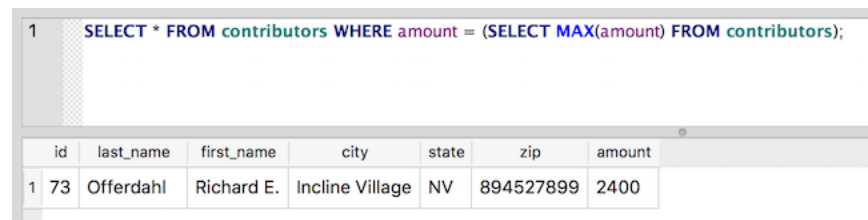
..which returns 2400.

```
SELECT * FROM contributors WHERE amount = 2400;
```

While that would work, it's a little clunky and brittle: If the database is being updated often, we'd always have to run the lookup for `MAX()` first, in case the maximum amount changed between queries.

Wouldn't it be nice to be able to combine those two queries into one statement? Well, we're in luck - a subquery is up to that task:

```
SELECT * FROM contributors WHERE amount = (SELECT MAX(amount) FROM contributors);
```



The screenshot shows a SQL query editor with a single line of code: `SELECT * FROM contributors WHERE amount = (SELECT MAX(amount) FROM contributors);`. Below the query, a table with columns `id`, `last_name`, `first_name`, `city`, `state`, `zip`, and `amount` is visible. The table contains one row: `1`, `73`, `Offerdahl`, `Richard E.`, `Incline Village`, `NV`, `894527899`, `2400`.

The subquery appears in parentheses, and it stands in for the value we want to test against `amount`. The subquery is executed first, and its result is used in the outer query. Because the subquery returns 2400, the query above gives the same result as a query for `amount = 2400`.

This statement works because our subquery only returns a single value (the value of `MAX(amount)`). It's also possible to use a subquery that returns multiple results, but in that case, we can't use the `=` operator.

If we wanted, for example, to get the total contributions from the top 20 contributors, we would have a list of 20 rows we want to match against. That's where our new friend `IN` comes to the rescue:



Note that we're spreading the query across multiple lines since the query statement is starting to get long. Formatting SQL statements in this way helps with readability as you start writing increasingly complex queries.

```
SELECT SUM(amount)
FROM contributors
WHERE id IN (
    SELECT id FROM contributors ORDER BY amount DESC LIMIT 20
);
```

The subquery returns the ids of the first 20 rows ordered by amount. The outer query asks for the sum of all amounts where the unique identifier for our contributor is in the results of our subquery. When we put them together, we get the sum of the amounts for the top 20 contributors:

The screenshot shows a SQL query in a text editor with line numbers 1 through 6. The query is:
   
1 SELECT SUM(amount)
   
2 FROM contributors
   
3 WHERE id IN (
   
4 SELECT id FROM contributors ORDER BY amount DESC LIMIT 20
   
5 );
   
6
   
Below the editor, a result table is displayed with one column labeled 'SUM(amount)' and one row containing the value '44500'.

|   | SUM(amount) |
|---|-------------|
| 1 | 44500       |

Note that there are other contributors in the list who have also donated 2100 (the smallest amount in the top 20), so the cut-off point is arbitrary. Depending on the story, we might want to do something more sophisticated with this query, such as looking for the sum of all amounts less than 500, or something even more ambitious, such as looking for the sum of all amounts within a certain percentile.

Subqueries can also be used with *DELETE*, *UPDATE* and *INSERT* statements.

## 2.11 GROUP BY

With some aggregate functions in our tool belt, we're ready to take advantage of one of SQL's more powerful features: *GROUP BY*. The *GROUP BY* statement is used in conjunction with aggregate functions to group the results by a given column. Doing so allows us to write queries that return *counts*, *sums*, *averages*, *minimums* and *maximums* per group.

For Excel users, this feature mirrors the functionality of PivotTables.

So, what is the total amount of contributions per state?

```
SELECT state, SUM(amount)
FROM contributors
GROUP BY state;
```

```

1 SELECT state, SUM(amount)
2 FROM contributors
3 GROUP BY state;

```

|   | state | SUM(amount) |
|---|-------|-------------|
| 1 | AL    | 2125        |
| 2 | AR    | 500         |
| 3 | AZ    | 2350        |
| 4 | CA    | 18460       |
| 5 | CO    | 2100        |

It's also possible to group by a combination of columns. So, we can get totals by city and state, as well:

```

SELECT city, state, SUM(amount)
FROM contributors
GROUP BY city, state;

```

```

1 SELECT city, state, SUM(amount)
2 FROM contributors
3 GROUP BY city, state;

```

|   | city       | state | SUM(amount) |
|---|------------|-------|-------------|
| 1 | Alamogordo | NM    | 1000        |
| 2 | Atlanta    | GA    | 1900        |
| 3 | Austin     | TX    | 3300        |
| 4 | Bastrop    | TX    | 250         |

And we can use the aggregate function in an *ORDER BY* statement to sort the results by total amount:

```

SELECT city, state, SUM(amount)
FROM contributors
GROUP BY city, state
ORDER BY SUM(amount) DESC;

```

```

1 SELECT city, state, SUM(amount)
2 FROM contributors
3 GROUP BY city, state
4 ORDER BY SUM(amount) DESC;
5

```

|   | city           | state | SUM(amount) |
|---|----------------|-------|-------------|
| 1 | Columbia Falls | MT    | 4600        |
| 2 | George West    | TX    | 4600        |
| 3 | Penn Valley    | CA    | 4600        |
| 4 | Downey         | CA    | 4200        |

The syntax of this last statement is a little tricky. The columns to group by are separated by commas, but there is no comma before `ORDER BY` or `DESC`.

Most relational database management systems require that every non-aggregate field in the `SELECT` statement also be included in the `GROUP BY` statement<sup>1</sup>. Because `SUM(amount)` is an aggregate, we can include it in the `SELECT` statement, even though it isn't included in the `GROUP BY` list. But if we want to include `city` in the `SELECT`, we should also include it in the `GROUP BY` as well.

## 2.12 HAVING

Now that we understand *grouping* and *aggregates*, let's try filtering the results based on an aggregate. To start, let's find all cities for which contributions total more than \$3,000. Here's a first stab at the query:

```
SELECT city, state, SUM(amount)
FROM contributors
WHERE SUM(amount) >= 3000
GROUP BY city, state
ORDER BY SUM(amount) DESC;
```

And ... no.

```
misuse of aggregate: SUM(): SELECT city, state, SUM(amount)
FROM contributors
WHERE SUM(amount) >= 3000
GROUP BY city, state
ORDER BY SUM(amount) DESC;
```

The error message isn't very helpful, but you can see "misuse of aggregate: SUM()" is mentioned.

Turns out that aggregate functions can't be used in a `WHERE` clause. The `WHERE` clause acts as a filter on each row in turn, but here we want to test an expression against an aggregate value for a group of rows (`SUM(amount)`).

The equivalent of a `WHERE` clause for aggregates is `HAVING`. It appears after the `GROUP BY`:

```
SELECT city, state, SUM(amount)
FROM contributors
GROUP BY city, state
HAVING SUM(amount) >= 3000
ORDER BY SUM(amount) DESC;
```

<sup>1</sup> SQLite doesn't enforce this standard SQL restriction, which in some cases makes writing the query much simpler but in most cases can lead to unexpected results. But as a general practice and to make your queries portable to other systems, you should always include all columns for the `SELECT` in the `GROUP BY` list. If including that column in the `GROUP BY` isn't possible, then you'll probably need to use a *subquery* to create the desired result.

```
1  SELECT city, state, SUM(amount)
2  FROM contributors
3  GROUP BY city, state
4  HAVING SUM(amount) >= 3000
5  ORDER BY SUM(amount) DESC;
```

|   | city           | state | SUM(amount) |
|---|----------------|-------|-------------|
| 1 | Columbia Falls | MT    | 4600        |
| 2 | George West    | TX    | 4600        |
| 3 | Penn Valley    | CA    | 4600        |
| 4 | Downey         | CA    | 4200        |
| 5 | San Antonio    | FL    | 4200        |
| 6 | Wellsboro      | PA    | 4200        |
| 7 | Westfield      | NJ    | 4200        |
| 8 | Houston        | TX    | 3550        |
| 9 | Austin         | TX    | 3300        |

To get a better sense of the difference between `WHERE` and `HAVING`, let's first look at a fairly simple query using `WHERE`:

```
SELECT city, state, amount
FROM contributors
WHERE amount >= 2300;
```

This query looks for individual contributors who have given at least \$2,300, and it returns their city, state and amount.

```

1 SELECT city, state, amount
2 FROM contributors
3 WHERE amount >= 2300;

```

|   | city           | state | amount |
|---|----------------|-------|--------|
| 1 | Columbia Falls | MT    | 2300   |
| 2 | Columbia Falls | MT    | 2300   |
| 3 | Austin         | TX    | 2300   |
| 4 | Penn Valley    | CA    | 2300   |
| 5 | Penn Valley    | CA    | 2300   |

Now let's make this into an aggregate query by adding a `GROUP BY` and an aggregate function:

```

SELECT city, state, SUM(amount)
FROM contributors
WHERE amount >= 2300
GROUP BY city, state;

```

```

1 SELECT city, state, SUM(amount)
2 FROM contributors
3 WHERE amount >= 2300
4 GROUP BY city, state;

```

|   | city           | state | SUM(amount) |
|---|----------------|-------|-------------|
| 1 | Austin         | TX    | 2300        |
| 2 | Columbia Falls | MT    | 4600        |
| 3 | George West    | TX    | 4600        |
| 4 | Grafton        | NH    | 2300        |
| 5 | Houston        | TX    | 2300        |

We have the same nine cities that we had in the first query (those cities in which someone donated at least \$2,300). But now, rather than having one row per contributor, we have one row per city. The `GROUP BY` eliminates the duplicate

entries for cities in which more than one person contributed at least \$2,300. And by using the aggregate function for `SUM (amount)`, we're adding up all contributions of at least \$2,300 for each city.

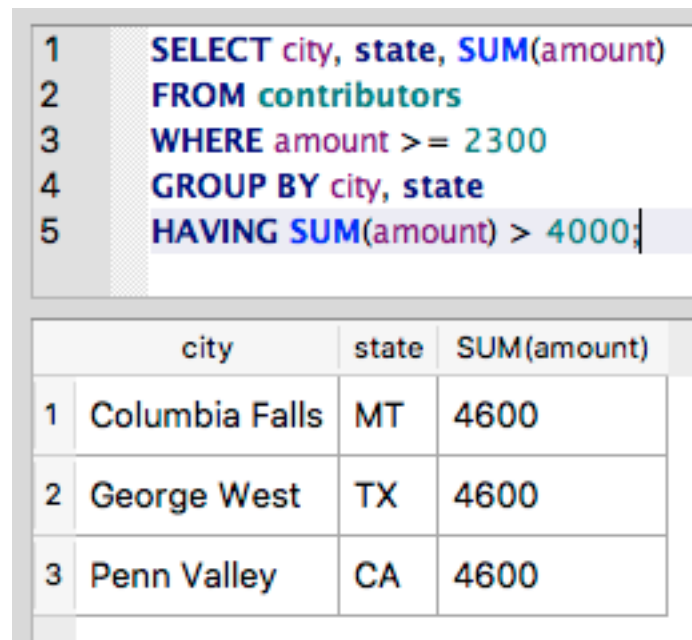
Now let's further filter this list of cities. We want to look only at cities in which these large contributions (\$2,300 or greater) made a big difference. Let's call \$4000 a big difference, for the sake of argument. So, we want only those cities for which the total amount of contributions at this size exceeds \$4000.

Looking at the results from the last query, we know to expect 3 rows, but it's not always so easy to see.

Here goes:

```
SELECT city, state, SUM(amount)
FROM contributors
WHERE amount >= 2300
GROUP BY city, state
HAVING SUM(amount) > 4000;
```

And bam! We now have a list of cities where large contributions totaled more than \$4000.



The image shows a screenshot of a SQL query editor. The query is as follows:

```
1 SELECT city, state, SUM(amount)
2 FROM contributors
3 WHERE amount >= 2300
4 GROUP BY city, state
5 HAVING SUM(amount) > 4000;
```

Below the query, the results are displayed in a table with three columns: city, state, and SUM(amount). The results are as follows:

|   | city           | state | SUM(amount) |
|---|----------------|-------|-------------|
| 1 | Columbia Falls | MT    | 4600        |
| 2 | George West    | TX    | 4600        |
| 3 | Penn Valley    | CA    | 4600        |

## 2.13 Revisiting subqueries

Before wrapping up Part II, let's revisit *subqueries*.

Recall that subqueries are SQL queries nested inside of a larger SQL statement. They're especially useful for dynamically filtering results on the fly as part of the `WHERE` clause. As we saw earlier, subqueries let us base the results of one query on the results of another, without having to run the queries separately.

But subqueries aren't limited to use in the `WHERE` clause. Another powerful – and perhaps surprising – use of subqueries is in `SELECT`.

For example, say that you wanted to determine the percentage of all contributions that came from each state.

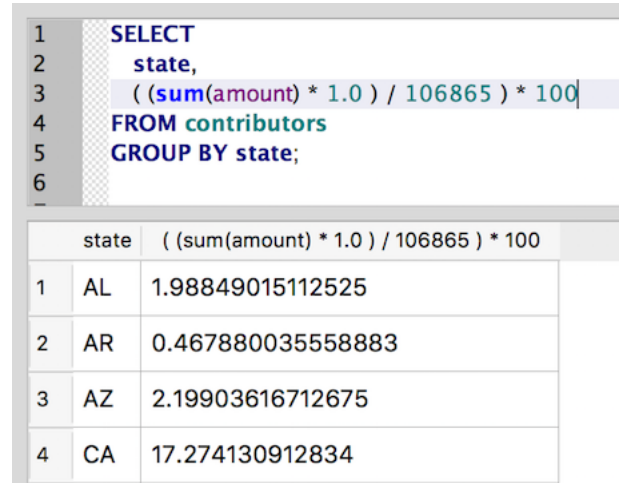
You *could* perform this operation with two separate queries, starting with a sum of all contributions:

```
select sum(amount) from contributors;
```

The above query gives us a total of \$106,865.

Next, we can use *GROUP BY* to sum contributions by state, and divide those totals by the sum of all contributions that we calculated above:

```
SELECT
    state,
    ( (sum(amount) * 1.0 ) / 106865 ) * 100
FROM contributors
GROUP BY state;
```



The screenshot shows a SQL editor with the following query:

```
1 SELECT
2   state,
3   ( (sum(amount) * 1.0 ) / 106865 ) * 100
4 FROM contributors
5 GROUP BY state;
6
```

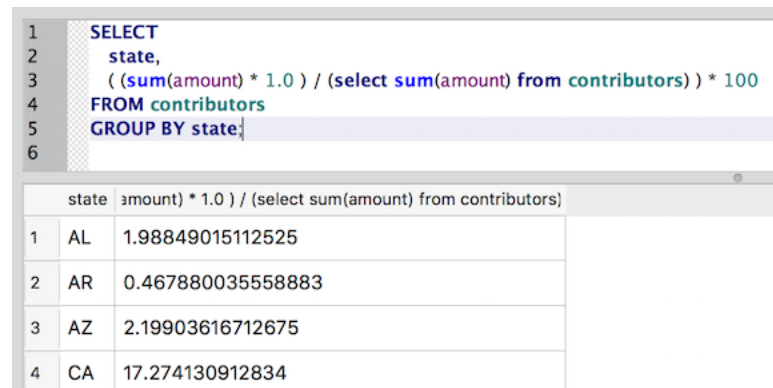
Below the query, a table displays the results:

|   | state | ( (sum(amount) * 1.0 ) / 106865 ) * 100 |
|---|-------|---|
| 1 | AL    | 1.98849015112525                        |
| 2 | AR    | 0.467880035558883                       |
| 3 | AZ    | 2.19903616712675                        |
| 4 | CA    | 17.274130912834                         |

This works, but wouldn't it be nice if we could dynamically calculate the sum of all contributions, rather than hard-code the total from the first query? That way, our calculation should "just work" if we add more contributions to the database.

This is where the *SELECT* subquery can work its magic:

```
SELECT
    state,
    ( (sum(amount) * 1.0 ) / (select sum(amount) from contributors) ) * 100
FROM contributors
GROUP BY state;
```



The screenshot shows a SQL editor with the following query:

```
1 SELECT
2   state,
3   ( (sum(amount) * 1.0 ) / (select sum(amount) from contributors) ) * 100
4 FROM contributors
5 GROUP BY state;
6
```

Below the query, a table displays the results:

|   | state | amount) * 1.0 ) / (select sum(amount) from contributors) |
|---|-------|--|
| 1 | AL    | 1.98849015112525   |
| 2 | AR    | 0.467880035558883  |
| 3 | AZ    | 2.19903616712675   |
| 4 | CA    | 17.274130912834  |

Above, we've simply replaced the hard-coded sum of all contributions with the query that generated the value. SQLite will calculate this total once and use it to determine each state's percentage of overall contributions.

Not too shabby. Subqueries in select statements can clearly be a powerful tool in your SQL skill set, especially when combined with aggregates, *GROUP BY* and other SQL features we've covered in Part II.

### 2.13.1 A word of caution

With this new power, of course, comes responsibility. As you begin writing increasingly complex queries, they will become harder to read – not to mention debug.

Be cautious as you craft such queries, making sure to format the SQL in a readable way. Execute subqueries independently before dropping them into a larger SQL query, to ensure they’re performing as expected. And for especially tricky syntax, add `code comments` to explain the logic.

## 2.14 Conclusion

So, now you can construct a vast array of query types in SQL. Using *subqueries*, *aggregates* and *GROUP BY*, you should be able to ask nearly anything of a single data set that you need.

In the next part, we’ll move on to exploring relationships between data sets, and you’ll be able to amaze your friends and colleagues with your raw SQL power.

See you in *Part III*.



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#).



## 3.1 Spreading the data around: Data Normalization

All of the queries we've run so far are limited to a single table. If all you ever do is import data from a spreadsheet, then you could always limit your queries to a single table. But most data projects of any depth soon involve multiple database tables.

Why would you want to separate the data into different tables? Well let's think back for a moment to the description of relational databases from Part I:

- Data is organized into tables (relations) that represent a collection of similar objects (e.g. contributors).
- The columns of the table represent the attributes that members of the collection share (last name, home address, amount of contribution).
- Each row in the table represents an individual member of the collection (one contributor).
- And the values in the row represent the attributes of that individual (Smith, 1228 Laurel St., \$250).

So, a table represents a set of similar objects, and the objects all share certain attributes. But we could stretch that definition quite a bit: Contributors all have addresses, but they also have recipients (the candidates who received the contributions). Should we include in our **contributors** table the candidate name, the campaign address and phone number, the office sought, the state in which the candidate is running, etc? What about the campaign treasurer's name? Committee positions the candidate holds? Previous offices held?

Including all of this loosely related data in a single table takes us pretty far afield of the original relation (Contributor). We would also be storing a lot of redundant data (all of the candidate data would be repeated for each contribution to a candidate). As a result, it could become difficult to update the data. Changing a candidate's address, for example, would require a change to each row containing a contributor to that candidate. In addition, it would become increasingly difficult to spot any data entry errors. Each misspelling of a candidate's name would be like adding a new candidate, and it would be easier to overlook the error amidst all the repeated data. Finally, all of this redundancy means we're taking up more disk space than needed. (This last isn't as big a concern as it once was when disk space was more expensive, but it can present problems.)

So, in order to help ensure [data integrity](#), to keep tables logically coherent and to reduce disk usage, most database designers implement some degree of [data normalization](#). There are varying degrees of normalization, known as the

“normal forms,” but for practical purposes the goal is to remove repetition and to keep only clearly related data in the same table.

So, let’s go back to our hypothetically bulky **contributors** table and do some minimal normalization. Let’s begin by imagining a table that looks like this:

| last name | first name | street                 | city       | state | zip   | amount | date       | candidate last name | candidate first name | candidate party |
|-----------|------------|------------------------|------------|-------|-------|--------|------------|---------------------|----------------------|-----------------|
| Ahrens    | Don        | 4034 Ren-nellwood Way  | Pleasanton | CA    | 94566 | 250.00 | 2007-05-16 | Huckabee            | Mike                 | R               |
| Agee      | Steven     | 549 Laurel Branch Road | Floyd      | VA    | 24091 | 500.00 | 2007-06-30 | Huckabee            | Mike                 | R               |

Even with only two sample rows, it’s easy to see the redundancy here. Any place we see repetition has potential for some normalization. Also, it’s fairly clear that the table really represents two different relations (contributors and candidates). So, one approach to restructuring this data is to create **contributors** and **candidates** tables and separate the data accordingly.

To get started, let’s create a fresh database. Start up *DB Browser for SQLite* and perform the following steps:

- Click the *New Database* button and create a database called **contributors\_candidates**.
- Save it somewhere you can find it, such as the Desktop.

Next, we’ll create the **candidates** table. Here’s the SQL CREATE TABLE statement to execute:




```
CREATE TABLE "candidates" (  
  "id" INTEGER PRIMARY KEY NOT NULL,  
  "first_name" TEXT NOT NULL,  
  "last_name" TEXT NOT NULL,  
  "middle_name" TEXT,  
  "party" TEXT NOT NULL  
);
```

This should all be old hat by now. We’re just creating a table for the candidates, including some basic information (name and party), and adding a **PRIMARY KEY**, a unique identifier for each candidate.

Now, let’s add some data to that table. Download `candidates.txt` and import it using the *File -> Import -> Table from CSV file...* menu (see [Importing Data](#) for more details).

- For the “Table name” field, the value should say *candidates*
- Make sure there’s a check mark in the “Column names in first line” box
- And set the “Field separator” value to Pipe (|).
- Click OK and when prompted, confirm that you want to import the data into the existing **candidates** table.

You should now have 17 rows in the **candidates** table:

Table:    

|    | id     | first_name  | last_name  | middle_name | party  |
|----|--------|-------------|------------|-------------|--------|
|    | Filter | Filter      | Filter     | Filter      | Filter |
| 1  | 16     | Mike        | Huckabee   |             | R      |
| 2  | 20     | Barack      | Obama      |             | D      |
| 3  | 22     | Rudolph     | Giuliani   |             | R      |
| 4  | 24     | Mike        | Gravel     |             | D      |
| 5  | 26     | John        | Edwards    |             | D      |
| 6  | 29     | Bill        | Richardson |             | D      |
| 7  | 30     | Duncan      | Hunter     |             | R      |
| 8  | 31     | Dennis      | Kucinich   |             | D      |
| 9  | 32     | Ron         | Paul       |             | R      |
| 10 | 33     | Joseph      | Biden      |             | D      |
| 11 | 34     | Hillary     | Clinton    | R.          | D      |
| 12 | 35     | Mitt        | Romney     |             | R      |
| 13 | 36     | Samuel      | Brownback  |             | R      |
| 14 | 37     | John        | McCain     |             | R      |
| 15 | 38     | Tom         | Tancredo   |             | R      |
| 16 | 39     | Christopher | Dodd       | J.          | D      |
| 17 | 41     | Fred        | Thompson   | D.          | R      |

So now, rather than having candidate data included with each row of the contributor data, we have one row for each candidate. It's a much cleaner data structure.



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](https://creativecommons.org/licenses/by-nc-nd/3.0/).

## 3.2 Referentially speaking: Associating tables using foreign keys

So, now we have the **contributors** table, but we also have a problem. Now that we've moved the candidate data out of the **contributors** table, how do we link contributors to their candidates? Without this link, we have no way of running queries that give, for example, total contributions per candidate. To create this reference between the two tables, we'll need a common field that the two tables share. The standard way of setting up this relationship is to include the **Primary Key** from the **referenced** table as a field in the **referencing** table. The new column in the referencing table is known as a **Foreign Key**.

Simply creating this foreign key column in the referencing table would be enough to let us run queries across both tables, but SQL also allows us to explicitly declare the foreign key and thus enforce this reference at the database level.

So, let's create a new **contributors** table, but in addition to the data about the contributor, let's add a **candidate\_id** field and let SQLite know that it is a foreign key referencing the **id** column in the **candidates** table:

```
CREATE TABLE "contributors" (
  "id" INTEGER PRIMARY KEY NOT NULL,
  "last_name" TEXT,
  "first_name" TEXT,
  "middle_name" TEXT,
  "street_1" TEXT,
```

(continues on next page)

(continued from previous page)

```

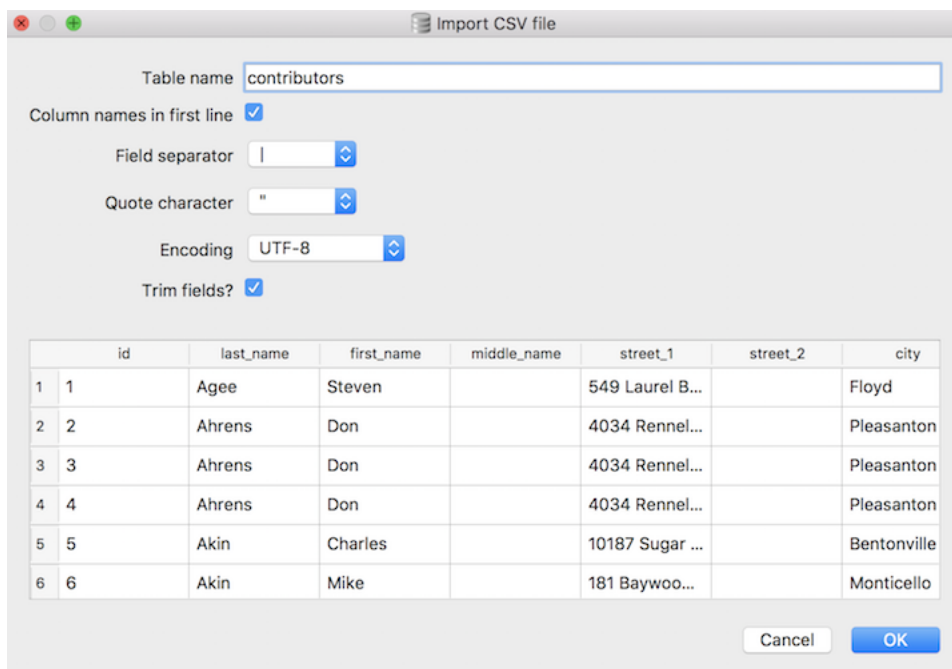
"street_2" TEXT,
"city" TEXT,
"state" TEXT,
"zip" TEXT,
"amount" INTEGER,
"date" TEXT,
"candidate_id" INTEGER NOT NULL,
FOREIGN KEY(candidate_id) REFERENCES candidates(id)
);

```

Notice the last two lines of that CREATE statement. The penultimate line adds the **candidate\_id** column, defines it as an integer, and makes it a required field (it cannot be null). The final line defines **candidate\_id** as a foreign key referencing the **id** column in the **candidates** table.

Now SQLite will enforce this reference, and if we try to enter a row in the **contributors** table without a **candidate\_id** or using a **candidate\_id** that doesn't actually appear in the **candidates** table, we'll get an error. In other words, every contributor must now have a candidate, and that candidate must already exist in the **candidates** table.

Now let's add some contributor data to the table. Download the text file at `contributors_with_candidate_id.txt` and import it into the **contributors** table using the *File -> Import -> Table from CSV file...* wizard:




- **NOTE:** You'll have to set the table name to **contributors**. Otherwise, SQLite will create a new table called **contributors\_with\_candidate\_id**, based on the name of the text file.
- Make sure there's a check mark in the "Column names in first line" box
- And set the "Field separator" value to Pipe (|).
- Click OK and when prompted, confirm that you want to import the data into the existing **contributors** table.

You should now have 175 rows in the **contributors** table<sup>1</sup>:

Table: contributors New Record Delete Record

|   | id     | last_name | first_name | middle_name | street_1           | street_2 | city        | state  | zip    |
|---|--------|-----------|------------|-------------|--------------------|----------|-------------|--------|--------|
|   | Filter | Filter    | Filter     | Filter      | Filter             | Filter   | Filter      | Filter | Filter |
| 1 | 1      | Agee      | Steven     |             | 549 Laurel Bran... |          | Floyd       | VA     | 24091  |
| 2 | 2      | Ahrens    | Don        |             | 4034 Rennellwo...  |          | Pleasanton  | CA     | 94566  |
| 3 | 3      | Ahrens    | Don        |             | 4034 Rennellwo...  |          | Pleasanton  | CA     | 94566  |
| 4 | 4      | Ahrens    | Don        |             | 4034 Rennellwo...  |          | Pleasanton  | CA     | 94566  |
| 5 | 5      | Akin      | Charles    |             | 10187 Sugar Cr...  |          | Bentonville | AR     | 72712  |
| 6 | 6      | Akin      | Mike       |             | 181 Bavwood L...   |          | Monticello  | AR     | 71655  |



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#).

### 3.3 Reaching across the aisle using JOIN

One way to run a query that uses data from two different tables is to use a *subquery*.

For example, to find all of the contributors to Barack Obama, you can do something like this:

```
SELECT *
FROM contributors
WHERE candidate_id = (
    SELECT id
    FROM candidates
    WHERE last_name = 'Obama' AND first_name = 'Barack'
);
```

This approach works fine as long as you're simply looking up values in one table and using them in the conditions for the `WHERE` clause. But often the queries you'll want to run will need to treat the two tables as a combined data set. A query that combines the data from two tables is known as a **join** on the tables. It is possible to do an **implicit** join simply by defining the relationship between the two tables in the `WHERE` clause:

```
SELECT contributors.last_name,
       contributors.first_name,
       candidates.last_name
FROM contributors, candidates
WHERE contributors.candidate_id = candidates.id;
```

<sup>1</sup> A quick aside about the text file: It contains a pre-populated `id` column, so we'll have unique **Primary Key** values. For the **candidates** table, we specified the `id` field in each row so that they would match the `candidate_id` values in this data. In a real project, we would probably use autoincrementing values for the ids in the **candidates** table, and populating the `candidate_id` field in the **contributors** table with the appropriate value would be a separate task.

```

1 SELECT contributors.last_name,
2    contributors.first_name,
3    candidates.last_name
4 FROM contributors, candidates
5 WHERE contributors.candidate_id = candidates.id;

```

|   | last_name | first_name | last_name |
|---|-----------|------------|-----------|
| 1 | Agee      | Steven     | Huckabee  |
| 2 | Ahrens    | Don        | Huckabee  |
| 3 | Ahrens    | Don        | Huckabee  |
| 4 | Ahrens    | Don        | Huckabee  |

Notice that we’re including both of the tables in the `FROM` clause. Also notice that we’re using a fully-qualified version of the column names: `contributors.last_name`, `candidates.last_name`. We’re including the table name here because `last_name` appears in both tables. So, just using `last_name`, as we usually would, would be ambiguous (the last name of the contributor or the last name of the candidate?). Adding the table name and a dot (.) before the column name disambiguates the column.

### 3.3.1 Using Aliases

Including the full table name with each column name can become a bit tedious. So, SQL allows you to define an *alias* for the table. To do so, simply include the alias after the table name in the `FROM` clause. Then you can use that alias, rather than the full table name, elsewhere in the query:

```

SELECT a.last_name, a.first_name, b.last_name
FROM contributors a, candidates b
WHERE a.candidate_id = b.id;

```

This query returns the same results as the one above, but it saves some typing by making “a” an alias for *contributors* and “b” an alias for *candidates*. The alias can use any valid table name you like, but obviously shorter aliases will save more typing, while longer ones may make the intention of the query easier to understand.



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#).

## 3.4 Explicit JOIN syntax

In addition to the *implicit join* syntax, SQL includes an explicit `JOIN` keyword. So, we could write the *earlier query* using this syntax instead:

```

SELECT contributors.last_name,
    contributors.first_name,
    candidates.last_name
FROM contributors
JOIN candidates ON contributors.candidate_id = candidates.id;

```

The query results should be the same as earlier, but using the `JOIN` keyword makes the intent of the query more explicit.

Aliases work with JOIN as well:

```
SELECT a.last_name,
       a.first_name,
       b.last_name
FROM contributors a
JOIN candidates b ON a.candidate_id = b.id;
```

Now let's try something a bit more interesting:

```
SELECT b.id, b.last_name, count(a.id)
FROM contributors a
JOIN candidates b ON a.candidate_id = b.id
GROUP BY b.id, b.last_name;
```

```
1  SELECT b.id, b.last_name, count(a.id)
2  FROM contributors a
3  JOIN candidates b ON a.candidate_id = b.id
4  GROUP BY b.id, b.last_name;
```

|   | id | last_name | count(a.id) |
|---|----|-----------|-------------|
| 1 | 16 | Huckabee  | 25          |
| 2 | 20 | Obama     | 25          |
| 3 | 22 | Giuliani  | 25          |
| 4 | 32 | Paul      | 25          |
| 5 | 34 | Clinton   | 25          |
| 6 | 35 | Romney    | 25          |
| 7 | 37 | McCain    | 25          |

Excellent! We now know that we have 25 contributors for each candidate. Very cool. But, hey, wait. Our list of candidates seems to be coming up short. Let's check it:

```
SELECT DISTINCT id, last_name FROM candidates;
```

The above query shows that we have 17 candidates total – in other words, the JOIN query is missing 10 candidates. What going on here? SQLite has gone mad!

Actually, there's a pretty sensible explanation for this result. We said earlier that performing the JOIN would return the same results as the query with this clause: WHERE contributors.candidate\_id = candidates.id.

What if a candidate has no contributors? Then that candidate is not returned by the query.

The JOIN acts just like the WHERE clause and filters out any rows that don't match the condition defined. Joins that return only rows in which there is a match in both tables are known as **INNER JOINS**. This is often exactly the behavior you want from the join (ignore any rows from either table that don't relate to a row in the other table). So by default, the JOIN keyword executes an INNER JOIN. You can also explicitly request an INNER JOIN, just to make things clearer:

```
SELECT b.id, b.last_name, count(a.id)
FROM contributors a
```

(continues on next page)

(continued from previous page)

```
INNER JOIN candidates b ON a.candidate_id = b.id
GROUP BY b.id, b.last_name;
```

The results will be the same.



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#).

## 3.5 OUTER JOIN

But how do we get the full list of candidates along with the number of contributors for each, including those candidates who have no contributors in our data set? SQL provides the “OUTER JOIN” syntax for doing just that. Outer joins are typically defined by the table from which we want to include non-matching rows, and we do so by referring to where that table appears in the JOIN statement.

- A `LEFT OUTER JOIN` includes all rows from the table on the left side of the statement and only matching rows from the table on the right side of the statement.
- A `RIGHT OUTER JOIN` includes all rows from the table on the right side of the statement and only matching rows from the left side of the statement.
- A `FULL OUTER JOIN` includes all rows from both tables.

Currently, SQLite only supports `LEFT OUTER JOIN` from the list above, but some other database management systems support the other two types as well.

---

**Note:** It’s easy to perform a `RIGHT OUTER JOIN` in SQLite by simply reversing the order of tables and using a `LEFT OUTER JOIN`. It’s also possible to do a `FULL OUTER JOIN` by combining `LEFT OUTER JOIN`s using the `UNION` keyword.

---

This all probably makes more sense in an example. Let’s rewrite the *grouping query* from earlier to include all candidates:

```
SELECT candidates.id,
       candidates.last_name,
       count(contributors.id)
FROM candidates
LEFT OUTER JOIN contributors ON candidates.id = contributors.candidate_id
GROUP BY candidates.id, candidates.last_name;
```

*(Aliases would work here as well, but I’ve used the full table names to make the relationships clearer.)*

Notice the JOIN statement: `candidates LEFT OUTER JOIN contributors`. Because **candidates** is on the left side of that statement, the result set will include all of the candidate rows, even those for which there are no matching **contributors**:



```

1  SELECT candidates.id,
2         candidates.last_name,
3         count(contributors.id)
4  FROM candidates
5  LEFT OUTER JOIN contributors ON candidates.id = contributors.candidate_id
6  GROUP BY candidates.id, candidates.last_name;

```

|   | id | last_name  | count(contributors.id) |
|---|----|------------|------------------------|
| 1 | 16 | Huckabee   | 25                     |
| 2 | 20 | Obama      | 25                     |
| 3 | 22 | Giuliani   | 25                     |
| 4 | 24 | Gravel     | 0                      |
| 5 | 26 | Edwards    | 0                      |
| 6 | 29 | Richardson | 0                      |
| 7 | 30 | Hunter     | 0                      |

Much better.



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#).

### 3.6 Why be normal? Denormalization as an informed choice.

Looking at the candidates table, there is another column showing some repetition: party. Many database designers would extract this column into its own table and then include a *party\_id* foreign key in the candidates table. It might be a good idea here to use that id rather than a text field; as it stands, if the data came in with “R,” “Republican” and “GOP” all appearing in that column, we would have a real mess. If we had a **parties** table that included only “R,” “D” and “I” (for independent), then we’d know we have a nonstandard value coming in when we tried to look up the *party\_id* for “GOP,” for example.

But normalization comes with a cost. Adding that parties table would mean that, any time we want to show candidate name and party, we’d have to do a join. And if we wanted contributor, candidate, and party, we’d have a query with two joins:

```

SELECT contributors.last_name,
       candidates.last_name,
       parties.name

```

(continues on next page)

(continued from previous page)

```
FROM contributors
JOIN candidates ON contributors.candidate_id = candidates.id
JOIN parties ON candidates.party_id = parties.id;
```

Doing multiple joins can become rather expensive in terms of memory, so often developers will create summary tables from the output of a `SELECT`:

```
CREATE TABLE contributors_candidates AS
  SELECT contributors.last_name,
         candidates.last_name,
         parties.name
FROM contributors
JOIN candidates ON contributors.candidate_id = candidates.id
JOIN parties ON candidates.party_id = parties.id;
```

But any changes to the **contributors** or **candidates** tables would immediately make this summary table out of date, so you'd have to create a way to update the summary table with each change.

There is another approach: **denormalization**. That is, collapsing your normalized data into a single table. If you're interested, check out the blog post on [codinghorror](#) and the spirited debate in the comments. I'll give Jeff Atwood the final comment here: "As the old adage goes, normalize until it hurts, denormalize until it works."



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#).

## 3.7 Conclusion

Congratulations! You now have a well-rounded set of SQL skills that can help you wrangle and analyze the most ornery of datasets. SQL JOINS in particular will help you design well-structured databases and join to other data sets in pursuit of more sophisticated analyses.

Below are some topics we have not yet covered that are worth exploring:

- The [UNION operator](#), which allows you to combine results from multiple queries
- Database optimization using [indexes](#)
- Database [views](#), which can be used to store complicated queries as a virtual table

Thanks for working through this tutorial! Please drop us a note on [Github](#) if you have thoughts on how the tutorial can be improved.



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#).

## 3.8 Further Resources

- <http://www.dbbm.fiocruz.br/class/Lecture/d17/sql/jhoffman/sqltut.html>

- <http://zetcode.com/databases/sqlitetutorial/>
- <http://www.sqlite.org/lang.html>
- [http://www.sqlite.org/lang\\_keywords.html](http://www.sqlite.org/lang_keywords.html)
- [http://www.sqlite.org/lang\\_expr.html](http://www.sqlite.org/lang_expr.html)
- <http://www.sqlite.org/foreignkeys.html>
- <http://en.wikipedia.org/wiki/SQL>



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#).



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#).



Some useful things that go beyond the more general realm of SQL.

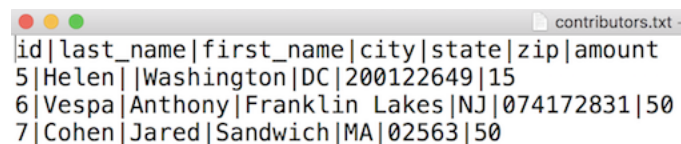
## 4.1 Importing data from a file

One common task we all face in data management is importing a data set into the database. Often, we receive a file in some other format such as MS Excel, CSV (comma-separated values) or tab-delimited and we want to get those values into a database table in order to run SQL queries on them.

Each database management system handles importing values from a file a bit differently. *DB Browser for SQLite* provides a nice interface for performing data imports from text files.

First, let's grab a plain text file full of contributors from the FEC database. Download `contributors.txt` and save it somewhere you can find it (your Desktop is a good place).

Check out the first few lines of the file below. Notice that this file is pipe-delimited (the columns are separated by the `|` character).



```
id|last_name|first_name|city|state|zip|amount
5|Helen||Washington|DC|200122649|15
6|Vespa|Anthony|Franklin Lakes|NJ|074172831|50
7|Cohen|Jared|Sandwich|MA|02563|50
```

I find this delimiter easy to use because it's unlikely to appear within a value in the import data. But using comma or tab characters to separate the values will work as well.

Now that we know what we're importing, let's try importing the data into the "contributors.db" we created in [Part 1](#) of the tutorial.

- Fire up *DB Browser*
- Click "Open Database"
- Locate your "contributors.db" file and click OK
- Start up the import wizard by selecting `File -> Import -> Table from CSV file...`

- Navigate to the “contributors.txt” file that you downloaded, and click Open.

The import wizard should appear, which you can use to define your import.

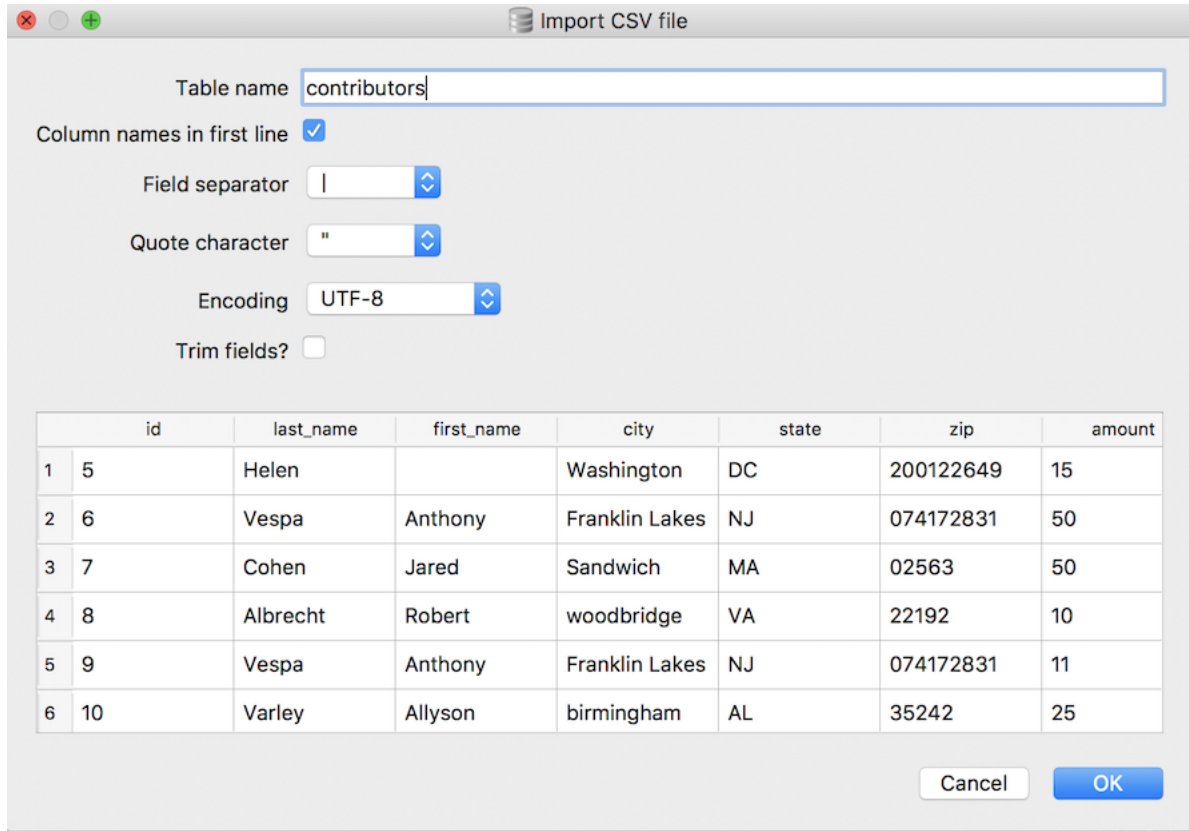


Table name: contributors

Column names in first line: ☒

Field separator: |

Quote character: "

Encoding: UTF-8

Trim fields?: ☐

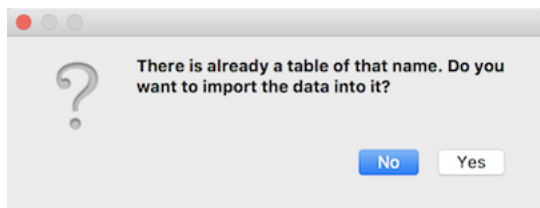
|   | id | last_name | first_name | city           | state | zip       | amount |
|---|----|-----------|------------|----------------|-------|-----------|--------|
| 1 | 5  | Helen     |            | Washington     | DC    | 200122649 | 15     |
| 2 | 6  | Vespa     | Anthony    | Franklin Lakes | NJ    | 074172831 | 50     |
| 3 | 7  | Cohen     | Jared      | Sandwich       | MA    | 02563     | 50     |
| 4 | 8  | Albrecht  | Robert     | woodbridge     | VA    | 22192     | 10     |
| 5 | 9  | Vespa     | Anthony    | Franklin Lakes | NJ    | 074172831 | 11     |
| 6 | 10 | Varley    | Allyson    | birmingham     | AL    | 35242     | 25     |

Cancel OK

- Check the “Column names in first line” checkbox.
- Select the **pipe(l)** for “Field separator”
- Uncheck the “Trim fields?” checkbox

For everything else, you can keep the default selections.

Click OK and you should get a pop-up notifying you that a table of the same name (*contributors*) already exists, and asking if you want to import the data into that table.



There is already a table of that name. Do you want to import the data into it?

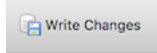
No Yes

Click “Yes” and the data import should proceed.

You should now have 103 rows of data to play with (the newly imported 100 rows, plus the original three add during the *Inserting Data* section).

The FEC data is dirty: there are missing fields, first names include middle names, there are weird values for some columns. Play around with it using the SQL you know, and see what you can find out. If you get surprising results from a query or are wondering how to do something, add a comment to the blog post.

You should also save the database changes you’ve made so far so you don’t lose your work. You can save the changes by clicking the “Write Changes” button:




## 4.2 Saving scripts

As you start writing more SQL, it’s helpful to work on each query in a separate SQL pane and to save your work as reusable scripts.


*DB Browser for SQLite* offers a few handy features for this workflow, including the ability to save and re-open scripts.

If you navigate to the *Execute SQL* panel, you should see a series of buttons towards the upper left that look like this:

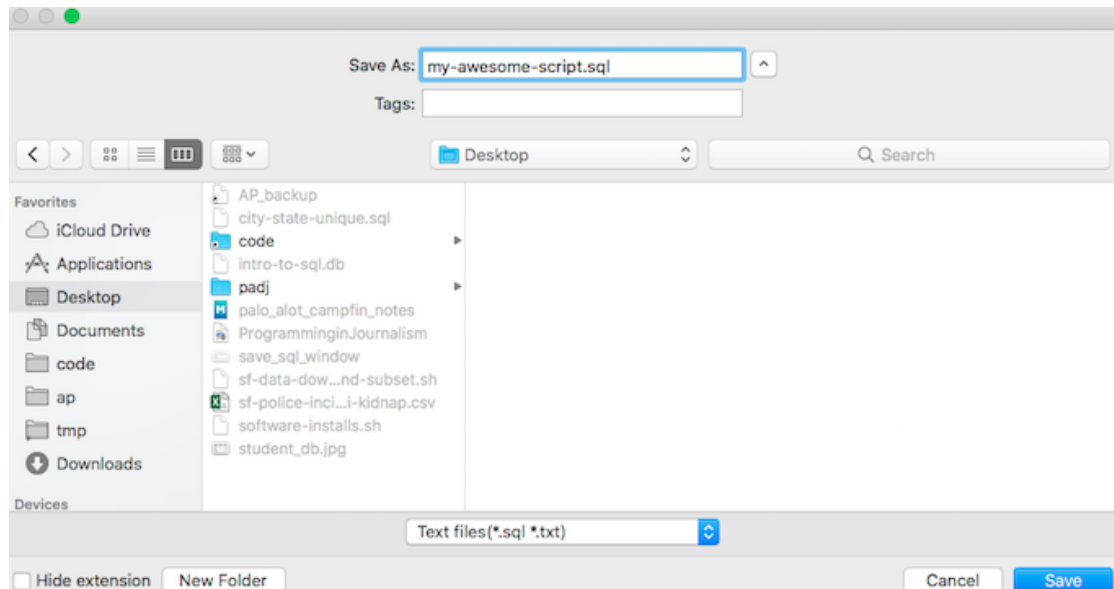



You can create new tabs for additional SQL queries by clicking the *Open tab* button (  ).



You can save the SQL in any tab using the *Save SQL file* button (  ). This will fire up a window that lets you choose a location and name for your script.

Scripts should always have a *.sql* file extension, e.g. *my-awesome-script.sql*



Finally, you can re-open a saved script by clicking the *Open SQL file* button (  ).



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#).



## CHAPTER 5

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



*A Gentle Introduction to SQL Using SQLite* by Troy Thibodeaux is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#).