RELATIONAL DATABASES / SQL

Relational databases are programs that store data, ultimately in files, but with additional data structures and interfaces that allow us to search and store data more efficiently

When working with data, we generally need four types of basic operations with the acronym CRUD:

- CREATE
- READ
- UPDATE
- DELETE

SQL

With another programming language, SQL **(pronounced like "sequel"), we can interact with databases with verbs like:**

- CREATE, INSERT
- SELECT
- UPDATE
- DELETE, DROP

Syntax in SQL might look like:

```
CREATE TABLE table (column type, ...);
```

With this statement, we can create a **table**, which is like a spreadsheet with rows and columns

In SQL, we choose the types of data that each column will store

We'll use a common database program called **SQLite**, one of many available programs that support SQL. Other database programs include Oracle Database, MySQL, PostgreSQL, and Microsoft Access

SQLite stores our data in a binary file, with 0s and 1s that represent data efficiently. We'll interact with our tables of data through a command-line program, sqlite3

We'll run some commands in VS Code to import our CSV file into a database:

```
$ sqlite3 favorites.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> .mode csv
sqlite> .import favorites.csv favorites
```

First, we'll run the sqlite3 program with favorites.db as the name of the file for our database

With **.import**, SQLite creates a table in our database with the data from our CSV file

Now, we'll see three files, including favorites.db

```
$ ls
favorites.csv  favorites.db  favorites.py
```

We can open our database file again, and check the schema, or design, of our new table with .schema

```
$ sqlite3 favorites.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE IF NOT EXISTS "favorites"(
  "Timestamp" TEXT,
  "title" TEXT,
  "genres" TEXT
);
```

We see that .import used the CREATE TABLE ... command to create a table called favorites, with column names automatically copied from the CSV's header row, and types for each of them assumed to be text.

We can select, or read data, with:

```
sqlite> SELECT title FROM favorites;
```

With a command in the format **SELECT columns FROM table;** we can read data from one or more columns

For example, we can write SELECT title, genre FROM favorites; to select both the title and genre

SQL supports many functions that we can use to count and summarize data:

- AVG
- COUNT
- DISTINCT
- LOWER
- MAX
- MIN
- UPPER

We can clean up our titles as before, converting them to uppercase and printing only the unique values

```
sqlite> SELECT DISTINCT(UPPER(title)) FROM shows;
```

We can also get a count of how many responses there are:

```
sqlite> SELECT COUNT(title) FROM favorites;
```

We can also add more phrases to our command:

- **WHERE**, adding a Boolean expression to filter our data
- **LIKE**, filtering responses more loosely
- **ORDER BY**
- **LIMIT**
- **GROUP BY**

We can limit the number of results:

```
sqlite> SELECT title FROM favorites LIMIT 10;
```

We can also look for titles matching a string:

```
sqlite> SELECT title FROM favorites WHERE title LIKE "%office%";
```

The % character is a placeholder for zero or more other characters, so SQL supports some pattern matching, though not it's not as powerful as regular expressions are

We can select just the count in our command:

```
sqlite> SELECT COUNT(title) FROM favorites WHERE title LIKE "%office%";
```

If we don't like a show, we can even delete it:

```
sqlite> SELECT COUNT(title) FROM favorites WHERE title LIKE "%friends%";
sqlite> DELETE FROM favorites WHERE title LIKE "%friends%";
sqlite> SELECT COUNT(title) FROM favorites WHERE title LIKE "%friends%";
```

With SQL, we can change our data more easily and quickly than with Python

We can update a specific row of data:

```
sqlite> SELECT title FROM favorites WHERE title = "Thevoffice";
sqlite> UPDATE favorites SET title = "The Office" WHERE title = "Thevoffice";
sqlite> SELECT title FROM favorites WHERE title = "Thevoffice";
```

Now, we've changed that row's value

We can change the values in multiple rows, too:

```
sqlite> SELECT genres FROM favorites WHERE title = "Game of Thrones";
sqlite> UPDATE favorites SET genres = "Action, Adventure, Drama, Fantasy, Thriller, War" WHERE title = "Game of Thrones";
sqlite> SELECT genres FROM favorites WHERE title = "Game of Thrones";
```

With **DELETE** and **DROP**, we can remove rows and even entire tables as well

And notice that in our commands, we've written SQL keywords in all caps, so they stand out more

There also isn't a built-in way to undo commands, so if we make a mistake we might have to build our database again!

Tables

We'll take a look at our schema again:

```
sqlite> .schema
CREATE TABLE IF NOT EXISTS "favorites"(
  "Timestamp" TEXT,
  "title" TEXT,
  "genres" TEXT
);
```

If we look at our values of genres, we see some redundancy:

```
sqlite> SELECT genres FROM favorites
```

And if we want to search for shows that are comedies, we have to search with not just SELECT title FROM favorites WHERE genre = "Comedy";, but also ... WHERE genre = "Comedy, Drama";, ... WHERE genre = "Comedy, News";, and so on

We can use the LIKE keyword again, but two genres, "Music" and "Musical", are similar enough for that to be problematic

We can actually write our own Python program that will use SQL to import our CSV data into *two* tables:

```
# Imports titles and genres from CSV into a SQLite database

import cs50
import csv

# Create database
open("favorites8.db", "w").close()
db = cs50.SQL("sqlite:///favorites8.db")

# Create tables
db.execute("CREATE TABLE shows (id INTEGER, title TEXT NOT NULL, PRIMARY KEY(id))")
db.execute("CREATE TABLE genres (show_id INTEGER, genre TEXT NOT NULL, FOREIGN KEY(show_id) REFERENCES shows(id))")

# Open CSV file
with open("favorites.csv", "r") as file:

    # Create DictReader
    reader = csv.DictReader(file)

    # Iterate over CSV file
    for row in reader:

        # Canoncalize title
        title = row["title"].strip().upper()

        # Insert title
        show_id = db.execute("INSERT INTO shows (title) VALUES(?)", title)

        # Insert genres
        for genre in row["genres"].split(", "):

            db.execute("INSERT INTO genres (show_id, genre) VALUES(?, ?)", show_id, genre)
```

- First, we import the Python `cs50` library so we can run SQL commands more easily.
- Then, the rest of this code will import each row of `favorites.csv`.

Now, our database will have this design:

```
$ sqlite3 favorites8.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE shows (id INTEGER, title TEXT NOT NULL, PRIMARY KEY(id));
CREATE TABLE genres (show_id INTEGER, genre TEXT NOT NULL, FOREIGN KEY(show_id) REFERENCES shows(id));
```

We have one table, shows, with an id column and a title column. We can specify that a title isn't null, and that id is the column we want to use as a primary key.

Then, we'll have a table called genres, where we have a show_id column that references our shows table, along with a genre column

This is an example of a **relation**, like a link, between rows in different tables in our database

In our shows table, we'll see each show with an id number:

```
sqlite> SELECT * FROM shows;
```

And we can see that the genres table has one or more rows for each show_id:

```
sqlite> SELECT * FROM genres;
```

Since each show may have more than one genre, we can have more than one row per show in our genres table, known as a **one-to-many** relationship

Furthermore, the data is now cleaner, since each genre name is in its own row

We can select all the shows are that comedies by selecting from the genres table first, and then looking for those ids in the shows table:

```
sqlite> SELECT title FROM shows WHERE id IN (SELECT show_id FROM genres WHERE genre = "Comedy");
```

Notice that we've nested two queries, where the inner one returns a list of show ids, and the outer one uses those to select the titles of shows that match

Now we can sort and show just the unique titles by adding to our command:

```
sqlite> SELECT DISTINCT(title) FROM shows WHERE id IN (SELECT show_id FROM genres WHERE genre = "Comedy")
ORDER BY title;
```

And we can add new data to each table, in order to add another show. First, we'll add a new row to the `shows` table for Seinfeld:

```
sqlite> INSERT INTO shows (title) VALUES("Seinfeld");
```

Then, we can get our row's `id` by looking for it in the table:

```
sqlite> SELECT * FROM shows WHERE title = "Seinfeld";
```

We'll use that as the `show_id` to add a new row in the `genres` table:

```
sqlite> INSERT INTO genres (show_id, genre) VALUES(159, "Comedy");
```

Then, we'll use `UPDATE` to set the title to uppercase:

```
sqlite> UPDATE shows SET title = "SEINFELD" WHERE title = "Seinfeld";
```

Finally, we'll run the same command as before, and see our new show is indeed in the list of comedies:

```
sqlite> SELECT DISTINCT(title) FROM shows WHERE id IN (SELECT show_id FROM genres WHERE genre = "Comedy")
ORDER BY title;
```