Introduction to SQL

SQL Query Language

- **SQL** (sometimes pronounced **"Sequel"**) is a query language that allows applications to interact with databases by creating tables, inserting data, updating data, fetching, etc.
- SQL provides a precise syntax for working with the database that's also easy to write and understand.
- SQL is a standard for interacting with relational databases.

SQL Query Language

We've spent a lot of time learning how to design our data models, and SQL allows us to implement our models precisely as **schemas** that we can actually use with a DBMS for storing, updating, and retrieving data.

Each of the relations/tables in our model will be implemented as a **table** in the database, with constraints for data types, ranges, primary keys, etc. This is where our database goes from being an idea to a real thing!

SQLite DBMS

- In this class, we'll use SQLite, which is a popular SQL-based DBMS. According to their website, https://sqlite.org/, it's the most used DBMS in the world!
  - Note that even though SQL is standardized, different DBMSs may use slightly different syntax, so not all SQL code you find on the web will work with SQLite
- SQLite doesn't run on its own server like most DBMSs do, but uses files on the disk. It's very easy to set up and use.
- SQLite is already installed on the GitHub Codespace Blank template, so we don't even need to install it!

SQLite in Codespaces

Before we start writing SQL queries, there is a bit of setup we should do first:

- Create a new Blank Codespace.
- Install the extension **SQLite Viewer** by **Florian Klampfer.** This will allow you to easily view database files in the Codespace's VS Code interface.
- Download the file **longlist.db** from Brightspace and place it in your project folder. (In Codespaces, you can drag files from your local file system.) This is a database file with a single table containing books nominated for the Booker prize! [Attribution: cs50] Take a moment to view the data in this database! Experiment with the **SQLite Viewer** interface to see how it works.

SQLite in Codespaces

Start the **SQLite CLI** application from the terminal with the following command:

```
sqlite3
```

To see a list of commands that work in the CLI, run:

```
.help
```

To quit back to the command line, run the following command:

```
.q
```

(Make sure you start it again to continue with this lesson.)

SQLite in Codespaces

To open a database file:

.open name-of-file.db (use the actual name of the file.)

Go ahead and open the database file you downloaded from Brightspace! To see which database is open:

.databases

To see the names of the tables in the current database:

.ta

SQLite in Codespaces

To see the columns of a table, along with their data types:

.schema table_name (use the actual name of the table)

You can also see this information by looking at the file using **SQLite Viewer,** but sometimes it's nice to just quickly get it while you're querying.

To configure SQLite to show query results in a prettier format:

.mode box

SQL Queries

Okay, not that we know our way around the SQLite CLI, let's start writing some SQL queries! As you've probably noticed, commands to the SQLite program start with a dot **.**

However, SQL queries to the database do not.

The most basic SQL query is **SELECT** for getting data. Try this:

```
SELECT author, title FROM longlist;
```

SQL Queries

Okay, not that we know our way around the SQLite CLI, let's start writing some SQL queries! As you've probably noticed, commands to the SQLite program start with a dot **.**

However, SQL queries to the database do not.

The most basic SQL query is **SELECT** for getting data. Try this:

```
SELECT author, title FROM longlist;
```

The **SELECT** statement gets data from the database. The **FROM** statement is used to choose the **table** from which to get data.

SQL Queries

Okay, not that we know our way around the SQLite CLI, let's start writing some SQL queries! As you've probably noticed, commands to the SQLite program start with a dot **.**

However, SQL queries to the database do not.

The most basic SQL query is **SELECT** for getting data. Try this:

SELECT author, title FROM longlist;

The two identifiers after the **SELECT** statement, **author and title,** represent the **columns of data** we'd like to get from the table.

SQL Queries

Okay, not that we know our way around the SQLite CLI, let's start writing some SQL queries! As you've probably noticed, commands to the SQLite program start with a dot **.**

However, SQL queries to the database do not.

The most basic SQL query is **SELECT** for getting data. Try this:

```
SELECT author, title FROM longlist;
```

The **semicolon ;** at the end indicates the end of the query. If you omit it, it's understood that the query will continue on the line below.

SQL Queries

Okay, not that we know our way around the SQLite CLI, let's start writing some SQL queries! As you've probably noticed, commands to the SQLite program start with a dot **.**

However, SQL queries to the database do not.

The most basic SQL query is **SELECT** for getting data. Try this:

```
SELECT author, title FROM longlist;
```

Keywords are NOT case sensitive, but by convention they're written in CAPS. I strongly recommend writing this way for understandability. Note that you may NOT use reserved keywords for tables, columns, etc.

SQL Queries

The **WHERE** statement can be used to add conditions to your **SELECT** statement. For example, the following query ONLY gets books published by **Scribe UK**:

```
SELECT author, title, publisher FROM longlist
WHERE publisher='Scribe UK';
```

Notice that the value we're using for the condition, **'Scribe UK'** is contained in **single quotes**. This is important to indicate the data type is a String. Notice that **double quotes ""** will NOT work because double quotes are reserved for declaring identifier names. (More on this later.)

Creating Databases and Tables

Now that we've experimented with getting table from a pre-existing database, let's create our own. Exit SQLite CLI:

.q

And then create a new database like this:

sqlite3 test.db

SQLite has started the CLI and created a new database in memory, but you won't see the **test.db** file appear until you create a table.

Creating Databases and Tables

To create a table in your new database, run the following query (note that we're writing this on multiple lines):

```
CREATE TABLE langara_courses (
course_name TEXT,
number INTEGER );
```

Creating Databases and Tables

To create a table in your new database, run the following query (note that we're writing this on multiple lines):

```
CREATE TABLE langara_courses (
course_name TEXT,
number INTEGER );
```

This identifier represents the **name of the new table.**

Creating Databases and Tables

To create a table in your new database, run the following query (note that we're writing this on multiple lines):

```
CREATE TABLE langara_courses (
course_name TEXT,
number INTEGER );
```

These are the **names of the columns** in the new table.

Creating Databases and Tables

To create a table in your new database, run the following query (note that we're writing this on multiple lines):

```
CREATE TABLE langara_courses (
course_name TEXT,
number INTEGER );
```

These are the **data types** of the new columns. Later we'll learn more about how data types work in SQLite (it's a bit different than other DBMSs).

Creating Databases and Tables

To create a table in your new database, run the following query (note that we're writing this on multiple lines):

```
CREATE TABLE langara_courses (
course_name TEXT,
number INTEGER );
```

There are a lot of other features we can add to tables (such as constraints on data, primary and foreign keys, etc.) but we'll keep it simple for now.

Creating Databases and Tables

To create a table in your new database, run the following query (note that we're writing this on multiple lines):

```
CREATE TABLE langara_courses (
course_name TEXT,
number INTEGER );
```

Identifiers for table and column names are NOT case sensitive. So **Langara_Courses** is the same as **langara_courses**.

Creating Databases and Tables

To create a table in your new database, run the following query (note that we're writing this on multiple lines):

```
CREATE TABLE langara_courses (
course_name TEXT,
number INTEGER );
```

There are some rules about which identifier names you can use. They can't be the same as reserved KEYWORDS, and they should not contain special characters except underscore _

Creating Databases and Tables

To create a table in your new database, run the following query (note that we're writing this on multiple lines):

```
CREATE TABLE langara_courses (
"Course Name!"  TEXT,
number INTEGER );
```

If you really want to use case-sensitive names and other special characters, or spaces between words, you can enclose the name in **double quotes ""**.

Creating Databases and Tables

You should now see that your new database file has appeared in the file explorer. Let's add some rows of data to our new table, like this:

INSERT INTO langara_courses (course_name, number) VALUES

('Intro to Databases', 4921),

('Full-Stack Web Dev', 4936);

Creating Databases and Tables

You should now see that your new database file has appeared in the file explorer. Let's add some rows of data to our new table, like this:

```
INSERT INTO langara_courses (course_name, number) VALUES
('Intro to Databases', 4921),
('Full-Stack Web Dev', 4936);
```

Here, we have the **table name** and in () the **column names.**

Creating Databases and Tables

You should now see that your new database file has appeared in the file explorer. Let's add some rows of data to our new table, like this:

```
INSERT INTO langara_courses (course_name, number) VALUES
('Intro to Databases', 4921),
('Full-Stack Web Dev', 4936);
```

Here, we have the **rows** to be inserted, where the order of comma-separated values matches the order of the columns above. Remember that Strings must be enclosed in **single quotes**!

SQL Files

You've probably noticed that it's tedious to write all our queries in the command line, and maybe it would be nice to record a batch of queries in a file and run them all at once. Luckily, there's a way for us to do this using **.sql** files! Create a new file called **test.sql**, and place the following code:

```
-- Hello SQL! Comments are preceded by two dashes
SELECT author, title, rating FROM longlist WHERE rating > 4;
SELECT * FROM longlist WHERE author='Han Kang';
```

SQL Files

To run the file, start the SQLite CLI, open the **longlist.db** database, set **.mode box** and run the following command:

```
.read test.sql
```