Off Platform Project: Making a Database of Movies

Welcome! This is likely your first off-platform project in PostgreSQL. Because of this, the goals of this project are relatively straightforward: let's get comfortable using our PostgreSQL client (most likely Postbird) to interact with our PostgreSQL server! In this project, we'll focus on creating tables, inserting data into those tables, and running some basic queries to see that data. Let's get started!

Part 1 — Creating Tables

Let's begin by creating a table to store some of our favorite movies. Open up your Postgres client, connect to a database, and create a table named `films` that has a `TEXT` column to store the `name` of a film, and an `INTEGER` column to store a `release_year` for each film.

If you're using Postbird for your client, you can connect to a database by using the dropdown menu in the top left corner. You can then create a new table in two different ways:

1. You can go to the Query tab and write your `CREATE TABLE` statement as if you were writing SQL on Codecademy. Pressing "Run Query" will run your `CREATE TABLE` statement.
2. You can use the plus symbol in the bottom left corner of Postbird to create a new table. This will open a pop-up window that asks for the table name. After giving the table a name, you'll be directed to the "Structure" tab where you can use the interface to create more columns. Notice that when you create a new table using this method, some columns will be created by default (for us, a column named `id` was created by default). Many clients will have a way to use a GUI (graphical user interface) to do tasks that you could do yourself by writing lines of SQL. When using a GUI it's good practice to think about what lines of SQL the GUI is running behind the scenes.

Part 2 — Saving Our Favorite Movies

Let's go ahead and insert our favorite movies into our `films` table! We'll need to know the name and release year to create new entries.

Since you likely don't know all release dates off the top of your head, check out IMDb! It's a great place to search for and discover information about film industry trends and specific movies. Fun fact: IMDb is also using Postgres behind the scenes!

Some of our favorites at Codecademy include The Matrix (1999), Monster's Inc. (2001), and Call Me By Your Name (2017).

Write some `INSERT` statements to add some of your favorite movies to your table!

If you're working in Postbird, you can write your `INSERT` statements in the Query tab. Once again, clicking "Run Query" will run those statements. If you want to take a look at the values in your table without writing `SELECT` statements, you can take a look at the Content tab after running your `INSERT` statements.

Be careful not to add a movie multiple times to your table! If you do, you can write `DELETE` queries in the Query tab to remove rows from the table. Alternatively, you can use the GUI! You can do this by going to the Content tab, right-clicking on rows in your table, and selecting "Delete Row".

Part 3 — Browsing movies

We can use the Content tab to look at our table, but let's get some practice writing `SELECT` statements. Note that you can add a `WHERE` clause to filter your results a bit. For example, adding `WHERE release_year = 1999` to the end of your `SELECT` statement will find all of the movies released in `1999`.

Can you find all of the movies you entered to your table that were released the year you were born?

Take a look at how the results of a `SELECT` statement is displayed in the bottom half of the Postbird Query tab.

Finally, if you want to experiment with Postbird's GUI, try using the Content tab to do that same filter. You should see a search bar that lets you filter your table based on certain columns. Can you use the GUI to find all movies in your table released *after* the year you were born? Again, think about the SQL statements that the GUI is running behind the scenes.

Part 4 — Adding Supplementary Information

While the release year for movies is one helpful fact to store, having more information about a movie might help us better make a selection for movie night.

Some additional data we may want to store is:

- Runtime (in minutes)
- Category (e.g. Action, Comedy, Drama, etc.)
- Rating
- Box Office Earnings — note that for this column, we used the `BIGINT` data type because we wanted to store numbers in the billions!

There are many more!

Pick a few new attributes and add new columns to the `films` table. To do so, we'll need to make use of `ALTER TABLE` statements in the Query tab.

When you add a column, take a look at the Content tab. What values are entered in the existing rows for the new column?

Once again, you could use Postbird's GUI to take care of this for you! In the "Structure" tab, you can add new columns. When you use this GUI, notice that you might get an error if you keep "Allow null" unchecked. Again,

think about what SQL is being written behind the scenes depending on whether that box is checked or not.

Part 5 — Backfilling Data

After we added those new columns, you'll notice that the values for those columns are `NULL` for all rows that were already in the table. Populating missing data retroactively from when the table and initial data was originally inserted is often called "backfilling."

Let's spend some time adding this additional information to the rows already in our `films` table. To do so, you can use the Query tab to write `UPDATE` statements. Those statements will also use `SET` and `WHERE` clauses to set values of the new columns.

You can also use Postbird's GUI to fill in these missing values. If you go to the Content tab, you can double click on values to edit those values directly! Notice how the GUI will prevent you from adding data that doesn't meet the column's data type. For example, the GUI will prevent you from adding letters to an `INTEGER` column.

Part 6 — Adding Constraints

For our last task, let's add some constraints to our `films` table to ensure it's protected against entering improperly formatted data.

First, let's remind ourselves that data types are a form of constraint. We've seen this already when we tried to edit values in the table that didn't match the data types.

There are many other types of constraints beyond data types! Right now, it's possible for us to add two movies with the same name. Let's add a `UNIQUE` constraint to the `name` column so you can't add duplicate names (note that we realize this probably isn't a *great* constraint... There surely are many movies with the same name. A better unique constraint might be some `id` number. But for now, let's stick with `name`.)

We could have made the constraint when we first made the table, but now that the table is already created, we'll have to use an `ALTER TABLE` statement. Here's an example of adding a unique constraint to an already existing table — in this example, we're adding a constraint onto the `release_year` column (which isn't a very good idea to do!):

```
ALTER TABLE films
ADD CONSTRAINT unique_release UNIQUE (release_year);
```

Once you've added the constraint, take the time to experiment with it. Here are a few things that would be good to try:

- Look at the Structure tab to verify the existence of the constraint.
- Try violating the constraint. Try inserting a new movie with a duplicate name. What happens?
- Try updating a row's `name` so it is a duplicate with an existing row's name. What happens? You could try this either through writing SQL or through the GUI's Content tab.
- Try adding another constraint that is already violated by the data in the table. For example, what happens if you try to add a `UNIQUE` constraint to the `release_year` column, but there are already multiple movies with the `release_year` of `1999`?

Reflection

Nice work! By working on this project, you practiced writing SQL statements to create tables, insert data into those tables, edit that data, and add a few constraints. However, more importantly, you became more experienced with your PostgreSQL client. As our PostgreSQL projects become more complicated, it is important to be familiar with the tools that you're using. We encourage you to continue experimenting with the features of whatever PostgreSQL client you are using!