7.2.2

Create Tables in SQL

We've helped Bobby get more comfortable with pgAdmin, and now we have a database ready to go. Good thing we've already completed all of the other prep work with the data---we're ready to begin creating tables!

Using our final ERD as a guide, we'll create six tables, one for each CSV file. During the creation, we'll even map out the primary and foreign keys and assign data types. This will all be completed using pgAdmin's query editor, which is a code editor for SQL, much like VS Code is for Python.

These table creation statements will be our first introduction to Structured Query Language. A statement is a block of code that, when executed, sends a command to the database. Let's get started.

Review Diagrams

We have a database ready to go, so now we need to prep it for data. Take a quick look at our diagram to see what we need for the next step: recreating the same tables in the diagram in our SQL database.

Remember all the thought and care we put into creating the ERD? We took the extra time then because it's easier to iron out any mistakes or fix an error in a tool like Quick DBD than it is to recreate a table in Postgres.

Take a closer look at the Departments table from the diagram.

With the help of the diagram, we know the structure of this table: two columns with their data types. Also, the table is already named. All we need to do is transfer over the same information.

Looking back at the pgAdmin window, right-click on the database PH-EmployeeDB. Then, from the dropdown menu, scroll down to the Query Tool and click to select.
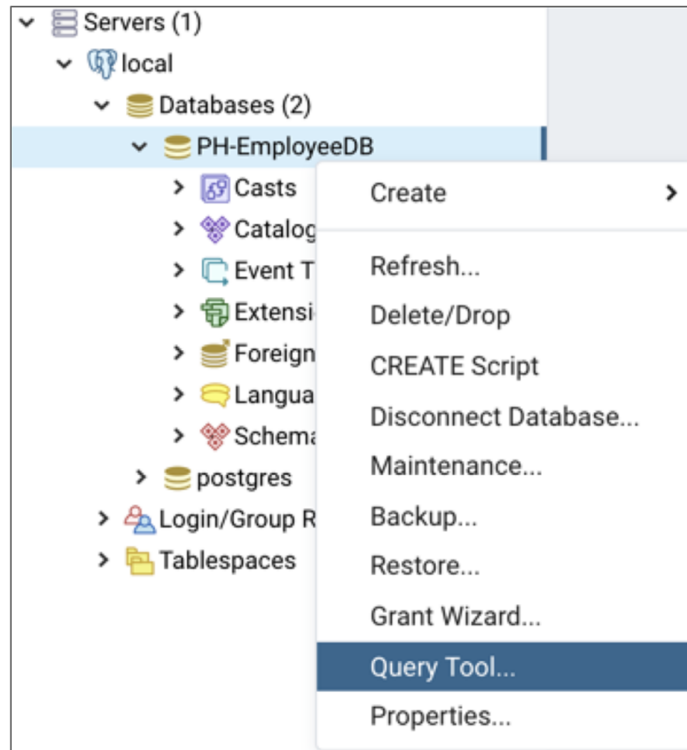
The Query Tool is pgAdmin's text editor, much like VSCode is for Python. This is where we will be helping Bobby construct all of the queries we'll need to perform analysis on the employee database.

With a brand new database, we can't just dive right in. There isn't any data available to query.

After opening the Query Tool, a query editor will appear in the pgAdmin window.

To create our new table, type the following code in the query editor:

```
-- Creating tables for PH-EmployeeDB
CREATE TABLE departments (
     dept_no VARCHAR(4) NOT NULL,
     dept_name VARCHAR(40) NOT NULL,
     PRIMARY KEY (dept_no),
     UNIQUE (dept_name)
);
```

Now we'll break down the code.

Write Statements to Create Tables

The very first line begins with a comment that clearly states the reason for the code. A SQL comment begins with two hyphens with no space between them: (--). Clear and concise comments add to the readability of any coding project

for yourself or any other developer who looks at it later. This one comment will apply to all of the `CREATE TABLE` statements, not just the one we are currently building.

Notice we're only adding one comment to capture all of the tables instead of one comment for each table. This is because these types of comments are a bit redundant; the code is already telling us what it's doing and doesn't need excess explanation.

> **NOTE**
>
> There is an ongoing discussion within the coding community regarding comments. One side of the argument is that comments aren't necessary—clean code is. However, there are many strong cases for adding comments.
>
> 1. Many employers appreciate code that clearly states what the code is for. Some companies even have developer style guides on writing comments.
>
> 2. Comments are often used as placeholders for future code. This helps developers keep track of their work.
>
> 3. They are also a way to add copyright and ownership information to proprietary code.
>
> There are many other reasons why comments are important, so getting used to strategically adding clean and clear comments is a good thing to start practicing early.

Break down the three main components of our `CREATE TABLE` statement.

```
-- Creating tables for PH-EmployeeDB
①CREATE TABLE②departments (
   ③ dept_no VARCHAR(4) NOT NULL,
```

1. `CREATE TABLE` is the syntax required to create a new table in SQL.

2. `departments` is the name of the table and how it will be referenced in queries.

   So the table has been named, now the structure needs to be created. The content inside the parentheses is how we'll do that.

3. `dept_no VARCHAR(4) NOT NULL,` creates a column named "dept_no" that can hold up to four varying characters, while `NOT NULL` tells SQL that no null fields will be allowed when importing data.

There are times when we don't want a data field to be null. For example, the dept_no column is our primary key—each row has a unique number associated with it. If we didn't have the `NOT NULL` constraint, then there's a chance that a row (or more than one row) won't have a primary key associated with the data.

Score to be Submitted   Based on last attempt

> Score
> Time spent: 0m 44s   Review   ◯ **100%**

                                        ⟳ Retake

Let's continue reviewing our code.

```
4 dept_name VARCHAR(40) NOT NULL,
5 PRIMARY KEY (dept_no),
6 UNIQUE (dept_name)
);
```

4. `dept_name VARCHAR(40) NOT NULL,` creates a column similar to the dept_no, only the varying character count has a maximum of 40.

5. `PRIMARY KEY (dept_no),` means that the dept_no column is used as the primary key for this table.

6. `UNIQUE (dept_name)` adds the unique constraint to the dept_name column.

IMPORTANT

A constraint is a rule that is applied to a column in a SQL table. It will limit the data in a way that provides more accuracy and reliability when working with it.

The unique constraint implies that the data in that column is unique. This ensures that if the table were to be updated in the future, nothing will be duplicated.

Take the table holding all of the department names and numbers, for example. Say that a developer wanted to update that table to add a new department, but reused department number d007.

If that were to happen, then d007 would reference current Sales Employees and Employees in the new department. Not a great situation, right? This creates dirty data, or data that is flawed, and would require extensive cleaning to fix.
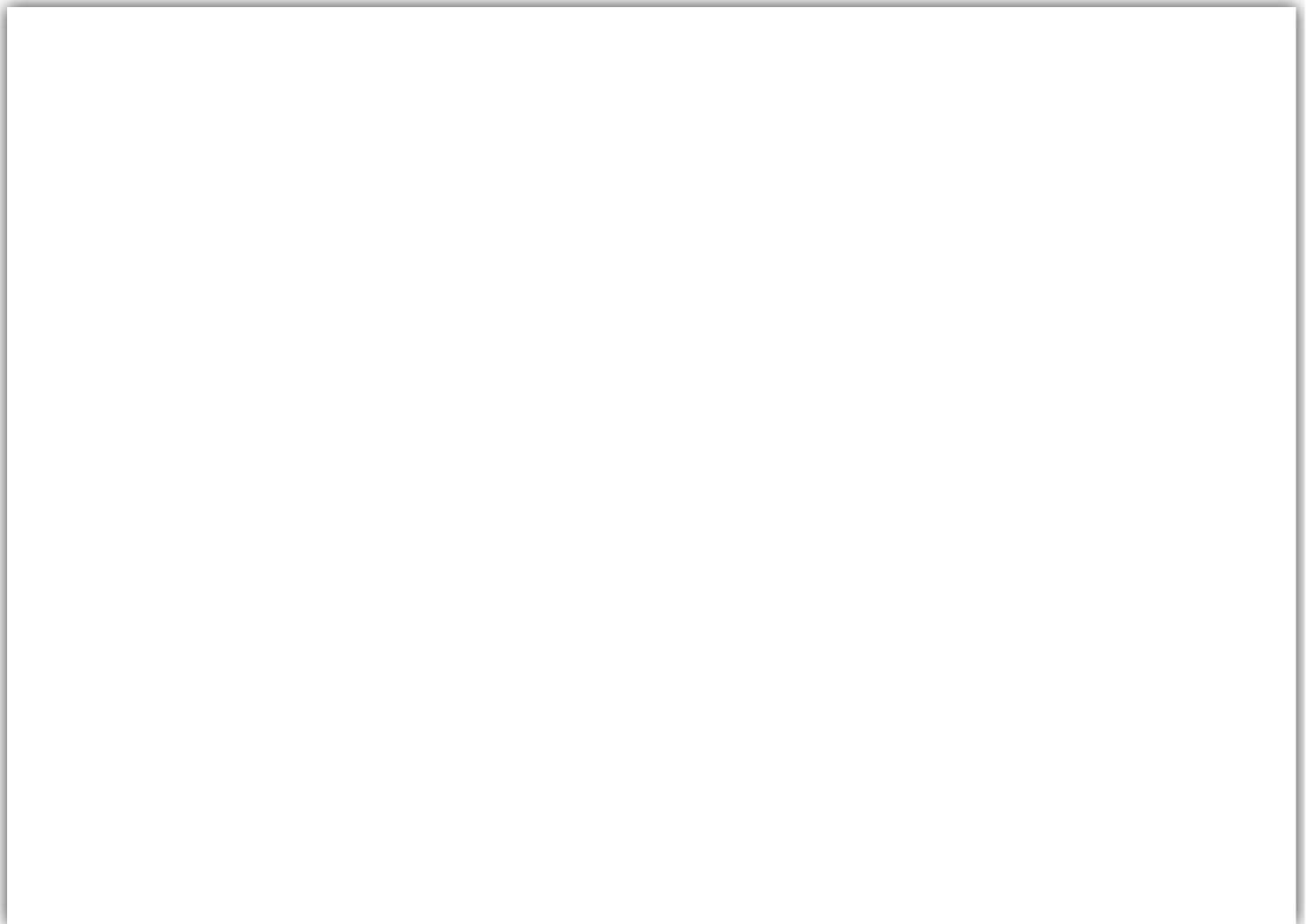
The closing parenthesis and semicolon signal that the SQL `CREATE TABLE` statement is complete. Any code added after will need to be included in a new SQL statement. A statement is a command that is set up with a certain syntax. Review our code again.

```
-- Creating tables for PH-EmployeeDB
CREATE TABLE  departments (
    dept_no VARCHAR(4) NOT NULL,
    dept_name VARCHAR(40) NOT NULL,
    PRIMARY KEY (dept_no),
    UNIQUE (dept_name)
);
```

Each statement will have the same syntax, or set of rules to follow, to perform a successful query. For instance, in the CREATE TABLE statement, we tell SQL we're creating a table, we name it, and then enclose any additional parameters within a set of parentheses.

The parameters within parentheses, such as the column names and primary key, are all indented. This is to help keep the code clean and readable.

A semicolon at the very end signals that the statement is complete. It's also part of readability in the syntax and if it's left out it may cause errors.

> **NOTE**
>
> Notice that the table names and column names are written in different cases. Table names are typically written in lowercase, while the rest of the statement is written in uppercase. This helps differentiate between table and column names as well as commands and parameters.

Execute the Code

To save the table to the database, we need to execute the code. To execute code in Python, we first save the `.py` file, then run it in the terminal. In Jupyter Notebook, code is executed by running each cell.

Because we are interacting directly with a database, pgAdmin executes code slightly differently. Right now we have a database, but nothing in it. Imagine that it is a blank canvas. The tables we create are the outline of the landscape, and later it will be filled with color (the data).

In the toolbar of the pgAdmin webpage, find and click the lightning bolt symbol toward the right of the bar. This button runs the code and saves our work to the database.

"Click this button" will run the CREATE TABLE statement and save the empty table to the database. A successful execution will return a message confirming the table creation.
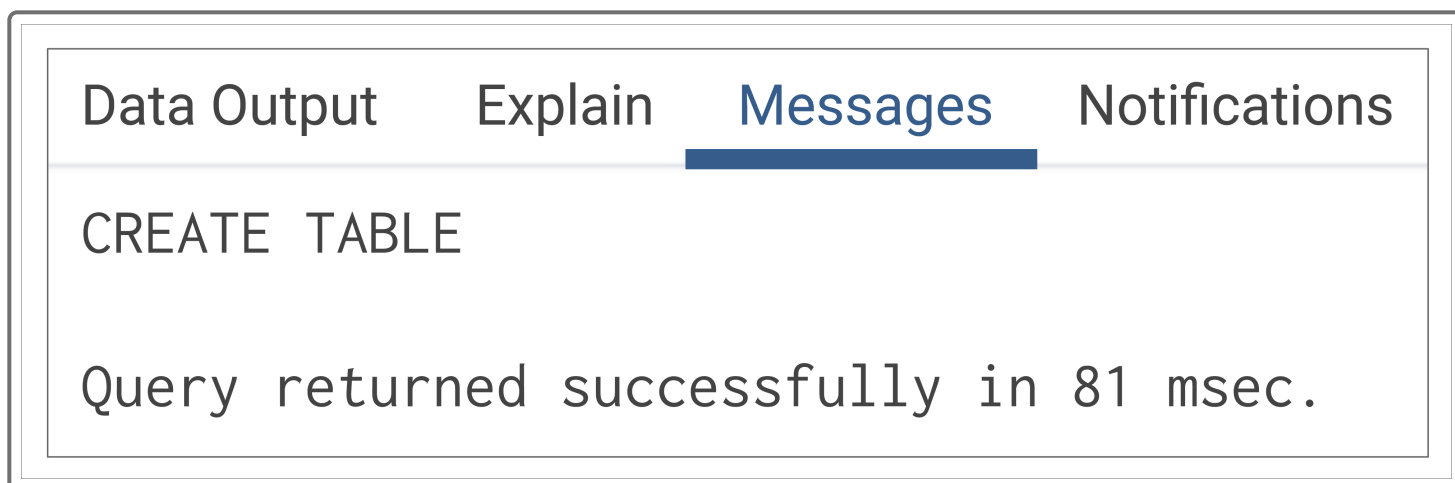
SHOW PRO TIP

Troubleshoot Error Messages

Any error message will also appear in the same manner. Encountering errors will happen often and troubleshooting them is a large part of being a developer. Thankfully, each error message encountered will tell us *why* the error occurred. This is great because it helps us, the developers, research and fix the problem.

Practice troubleshooting an error by executing the code to create the departments table again. After running the exact same code again, you'll get this error message:

```
ERROR: relation "departments" already exists
SQL state: 42P07
```

This error occurs because SQL data is persistent and cannot be overwritten if the same command is run again. Once a table has been committed to a database, it is there until a different command is run to delete it.

This helps preserve the integrity of the data already in place. Imagine adding a second Departments table with different data that overwrites the existing table.

IMPORTANT

Data integrity is the quality of the data we're working with. Clean data will yield better results in analysis, and maintaining the data integrity ensures greater accuracy and reliability.

Dirty data is data that contains errors such as duplicates, undefined values (i.e., not a number, or NaN), or other inconsistencies. This is why the `NOT NULL` constraint is in place.

To avoid encountering this error, highlight the code block you want to run first, then execute it. This tells pgAdmin to run *only* that code.

Sometimes error messages are more confusing and require additional research to solve. There's a good chance that lots of people have run into the same issue. Googling the error message itself will likely bring you to a conversation between developers about why the problem occurred and suggestions to fix it. We'll have more practice with troubleshooting as we work through the module.

Create Additional Tables

Create another table for Employees.

Employees

emp_no	🔑	int
birth_date		date
first_name		varchar
last_name		varchar
gender		varchar
hire_date		date

Just like last time, we know what we want to name the new table, what the columns are named, and the data type for each.

Start the new table with the `CREATE TABLE` statement and the name of the table.

```
CREATE TABLE employees (
```

REWIND

Like all coding languages, the ability to add comments is beneficial for developers. Comments are a place to add thoughts when working through a problem and to add explanation to code blocks. Adding clean and clear comments provides a roadmap to the code.

Don't forget the parentheses; remember, all of the details of the table will be inside them. If we make a mistake and the table isn't laid out to match the data, an error will occur during import, which will require troubleshooting—figuring out where the error originated and how to fix it.

NOTE

Troubleshooting is part of any coding language and it's something you will encounter from time to time. Learning how to address errors and implement fixes is a key part of being a developer, and we'll take time later to practice.

IMPORTANT

It's important to keep track of different data types in use. For example, if we use `int` instead of `varchar` for the first_name column, Postgres will search for integers instead of varying characters. Because the first name column will contain only letters and not the numbers pgAdmin is seeking, the import will generate an error and not be completed.

Continue with the table structure.

```
    emp_no INT NOT NULL,
    birth_date DATE NOT NULL,
    first_name VARCHAR NOT NULL,
    last_name VARCHAR NOT NULL,
    gender VARCHAR NOT NULL,
    hire_date DATE NOT NULL,
    PRIMARY KEY (emp_no)
);
```

Now we have everything needed to create a table: the SQL statement, the table components, and the closing parenthesis and semicolon.

Remember to execute the `CREATE TABLE` statement for the new table.

IMPORTANT

Did you run into an error when executing code the second time? That is because pgAdmin will run all of the code in the editor, unless told otherwise.

We have already created the first table and saved it to the database, so now we only want to save the second table:

1. Highlight only the code you want to run.

2. Click the lightning bolt (execute) button.

Create one more together—this time with foreign keys included. Add the following code to the bottom of your query
editor:

```
CREATE TABLE dept_manager (
dept_no VARCHAR(4) NOT NULL,
    emp_no INT NOT NULL,
    from_date DATE NOT NULL,
    to_date DATE NOT NULL,
FOREIGN KEY (emp_no) REFERENCES employees (emp_no),
FOREIGN KEY (dept_no) REFERENCES departments (dept_no),
    PRIMARY KEY (emp_no, dept_no)
);
```

It looks similar to the last table we created, except for two lines:

```
FOREIGN KEY (emp_no) REFERENCES employees (emp_no),
FOREIGN KEY (dept_no) REFERENCES departments (dept_no),
```

Remember that foreign keys reference the primary key of other tables. In the two lines above we can see that:

1. The `FOREIGN KEY` constraint tells Postgres that there is a link between two tables
2. The parentheses following `FOREIGN KEY` specify which of the current table's columns is linked to another
   table
3. `REFERENCES table_name (column_name)` tells Postgres which other table uses that column as a
   primary key

	References Current Table		Links to Second Table
FOREIGN KEY	(emp_no)	REFERENCES	employees (emp_no),
FOREIGN KEY	(dept_no)	REFERENCES	departments (dept_no),
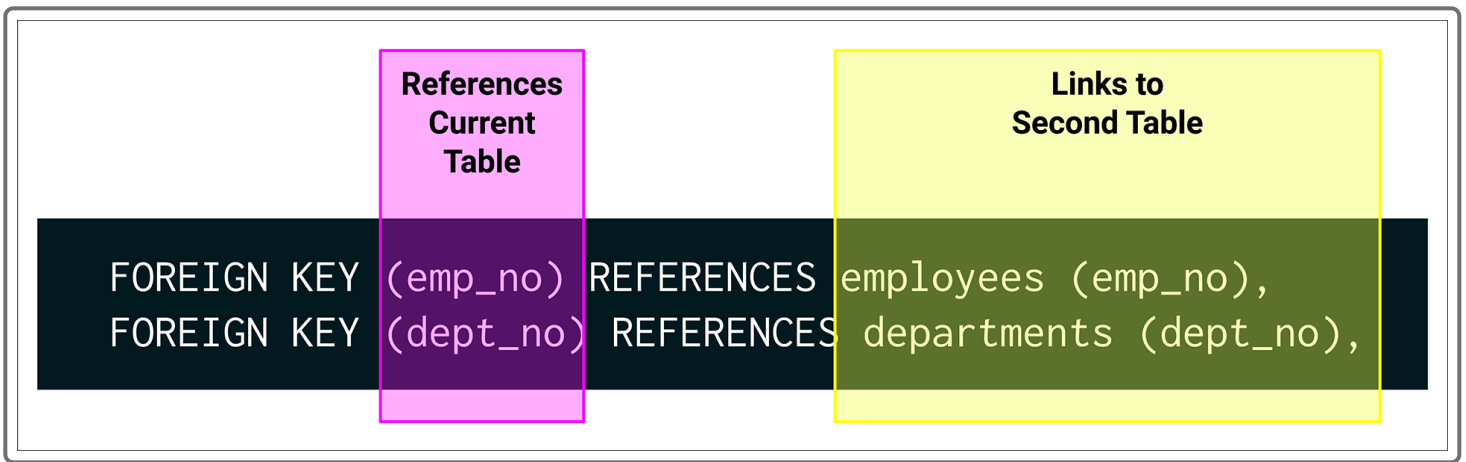
The primary key is similar, but there are two keys listed this time instead of just one. Remember the analogy of phone numbers being primary and foreign keys? Think of the two primary keys listed as two unique phone numbers that belong to dept_manager. The Employees and Departments tables are other numbers that have been saved as contacts. One thing to keep in mind when working with foreign keys is that it's possible that data insertion will fail if the foreign key isn't present. This is a "foreign key constraint" and in this case, it means that the new data needs a reference point (such as dept_no or emp_no) to be successfully added to the table.

Let's create another table for the data in `salaries.csv`. At the bottom of the query editor, type the following:

```sql
CREATE TABLE salaries (
  emp_no INT NOT NULL,
  salary INT NOT NULL,
  from_date DATE NOT NULL,
  to_date DATE NOT NULL,
  FOREIGN KEY (emp_no) REFERENCES employees (emp_no),
  PRIMARY KEY (emp_no)
);
```

This code tells Postgres that our new table is named "salaries" and we'll have columns for the emp_no, salary, from_date, and to_date. We also have specified that certain fields aren't allowed any null space with the `NOT NULL` constraint, which is important because we want this data to be persistent for every employee. As a final step in table creation, we've also specified primary and foreign keys.

SKILL DRILL

Using the entity relationship diagram as a guide, create the remaining tables in pgAdmin. Use the ERD as a reference guide if you need to.

Remember to use the REFERENCES keyword when creating the tables, such as we did when we created the dept_manager table. Otherwise, your code will generate errors when you import the data.

Query for Confirmation

Confirm the tables were created successfully by running a `SELECT` statement, which performs a query instead of constructing anything. In Python, we would create a script to tell a computer how to act. With SQL, we create queries instead. These queries are interactive—we're speaking directly to the data.

Think of it as asking the database a question. For example, say we want to know how many columns are in the departments table. How would we ask that particular question? We would create a SELECT statement, then run the code. This is called "querying the database."

In the editor, after the table creation statements, type `SELECT * FROM departments;`.

- The `SELECT` statement tells Postgres that we're about to query the database.
- The asterisk tells Postgres that we're looking for every column in a table.
- `FROM departments` tells pgAdmin which table to search.
- The semicolon signifies the completion of the query.

After executing the `SELECT` statement, pgAdmin will automatically show the result in the Data Output tab at the bottom of the page.

Data Output	Explain	Messages	Notifications
dept_no character varying (4)	dept_name character varying (40)		

Currently, we're able to see that the two columns exist and the type of data they contain. If we run the `SELECT` statement again after importing the CSV data, then the output would include the actual data as well. We'll start importing the data next.

IMPORTANT

The information in the database is static, which means that it will always be in the database unless directly altered, but the query editor is not. It's similar to working in a Microsoft Word document: If something happens to your computer before you saved your work, there's a good chance it'll be lost. If your computer crashes mid-query, the pgAdmin editor won't hold onto your code for you during a reboot.

Our queries are the meat and potatoes of SQL. We're finding connections between different tables and answering questions with the results. Even though losing a query isn't the end of the world (they can be rebuilt, after all), it does take time to replicate the work already completed.

To save your hard work in the query editor, it's actually a good idea to keep an external .sql file with your code saved in your class folder.

Copy and paste the code you used to create tables into a new VSCode window and save it as `schema.sql`. This way, you'll always have access to your code if you need to refer to or use it again.

Similarly, when we begin writing queries, create another `.sql` file named `queries.sql` and make sure to save successful queries in it as we go along.

ADD/COMMIT/PUSH

After saving your `schema.sql` file, remember to push it to your repo!

© 2020 - 2022 Trilogy Education Services, a 2U, Inc. brand. All Rights Reserved.