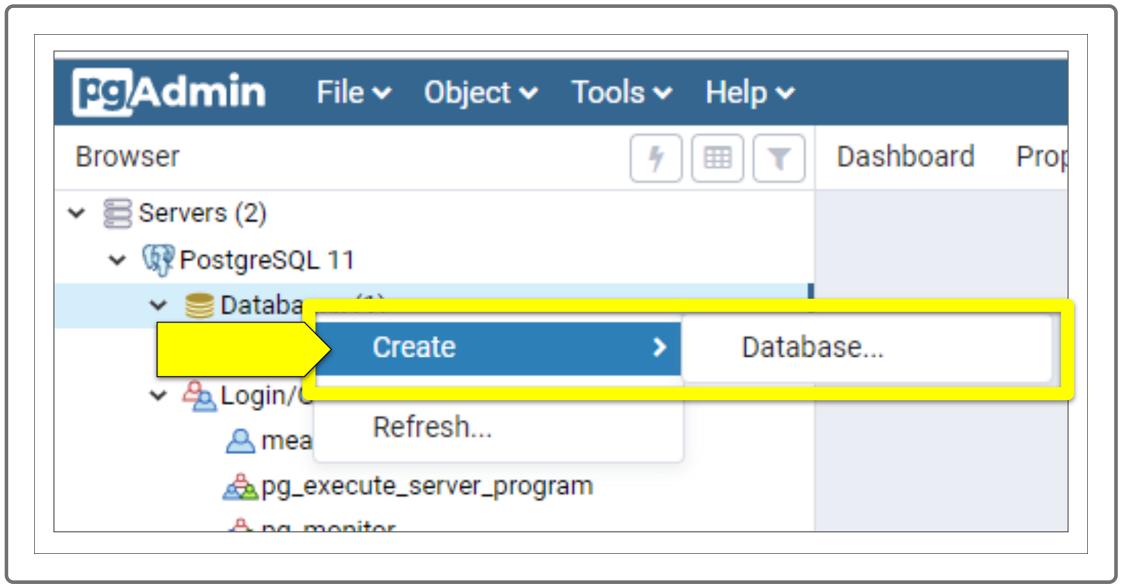
8.5.1 Connect Pandas and SQL

Amazing Prime has decided the easiest way to make the data accessible for the hackathon is to provide a SQL database to the participants. Britta needs to move the data from Pandas into a PostgreSQL database.

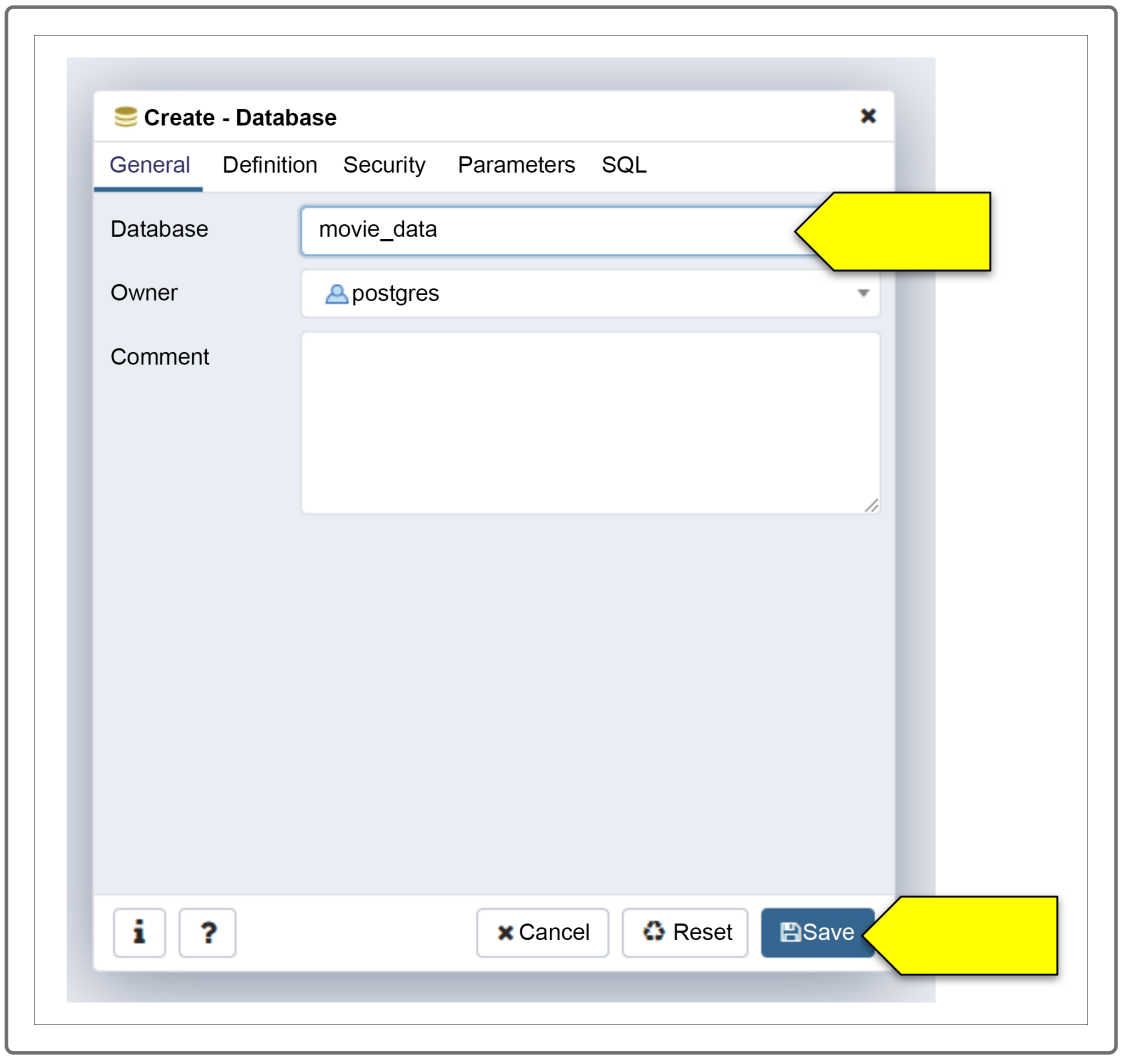
Now that we've extracted and transformed our data, it's time to load it into a SQL database. We're going to create a new database and use the built-in `to_sql()` method in Pandas to create a table for our merged movie data. We'll also import the raw ratings data into its own table.

Create a Database

Start pgAdmin and expand your local servers in the left-hand pane so you can see the Databases section. Right-click on Databases and select Create followed by Database.



Name the database "movie_data" and click Save.



Import Modules

Go back to your Jupyter Notebook. We need to import create_engine from the sqlalchemy module. Don't forget to add this import to the first cell so that all your imports are in one spot.

```
from sqlalchemy import create_engine
```

Now we can create the database engine that will allow Pandas to communicate with our SQL server.

Create the Database Engine

The database engine needs to know how to connect to the database. To do that, we make a connection string. For PostgreSQL, the connection string will look like the following:

```
"postgresql://[user]:[password]@[location]:[port]/[database]"
```

It looks similar to a website URL because it *is* a URL. The protocol here is "Postgres" instead of "http."

Unfortunately, this means that if we make just a simple string, we'll have to include our password, written in plaintext. Anyone who can see our code can get our database password and do anything they want with it.

REWIND

We're going to hide our database password in another file, and tell git to ignore it with `.gitignore`. Recall that we used this when we hid our API keys.

To be safe, we'll create a new file `config.py` that stores our password to a variable. We can add `config.py` to our `.gitignore` file so that git will not share our sensitive information.

Create the new file with Jupyter by going to New and selecting Text File. Rename the file `config.py`. In the text editor, add the following:

```
db_password = 'YOUR_PASSWORD_HERE'
```

In your Jupyter Notebook, add another import line. The standard practice is to import dependencies at the top, so add the following line to the top and run the cell again:

```
from config import db_password
```

For our local server, the connection string will be as follows:

```
db_string = f"postgresql://postgres:{db_password}@127.0.0.1:5432/movie_data"
```

This is all the information that SQLAlchemy needs to create a database engine.SQLAlchemy handles connections to different SQL databases and manages the conversion between data types. The way it handles all the communication and conversion is by creating a database engine.

Create the database engine with the following line in a new cell:

```
engine = create_engine(db_string)
```

IMPORTANT

> Some users may need an additional package installed for this section. In your terminal, run the following code: `pip install psycopg2-binary` to add it to your coding environment.

Import the Movie Data

To save the `movies_df` DataFrame to a SQL table, we only have to specify the name of the table and the engine in the `to_sql()` method.

```
movies_df.to_sql(name='movies', con=engine)
```

In pgAdmin, confirm that the table imported correctly. Follow these steps:

1. Right-click the "movies" table name and select Properties.
2. Click the Columns tab to make sure all columns have an appropriate data type.
3. Close the Properties window, and then right-click "movies" again.
4. Select "View/Edit Data" followed by "First 100 Rows."
5. Right-click "movies" and select Query Tool.
6. Inside the Query Editor, run the query `select count(*) from movies` to make sure all the rows were imported.

Nice work! Now it's time to import the ratings data.

Import the Ratings Data

The ratings data is too large to import in one statement, so it has to be divided into "chunks" of data. To do so, we'll need to reimport the CSV using the `chunksize=` parameter in `read_csv()`. This creates an iterable object, so we can make a `for` loop and append the chunks of data to the new rows to the target SQL table.

CAUTION

> The `to_sql()` method also has a `chunksize=` parameter, but that won't help us with memory concerns. The `chunksize=` parameter in `to_sql()` creates smaller transactions sent to SQL to prevent the SQL instance from getting locked up with a large transaction.

The simplest way to do this is with two lines:

```
# Do not run this yet!
for data in pd.read_csv(f'{file_dir}ratings.csv', chunksize=1000000):
    data.to_sql(name='ratings', con=engine, if_exists='append')
```

This can take quite a long time to run (more than an hour). It's a really good idea to print out some information about how it's running.

Let's add functionality to this code to print out:

- How many rows have been imported
- How much time has elapsed

Step 1: Print Number of Imported Rows

Below is the previous block of code, with comments added for refactoring:

```
# create a variable for the number of rows imported
for data in pd.read_csv(f'{file_dir}ratings.csv', chunksize=1000000):

    # print out the range of rows that are being imported

    data.to_sql(name='ratings', con=engine, if_exists='append')

    # increment the number of rows imported by the chunksize

    # print that the rows have finished importing
```

create a variable for the number of rows imported

We'll call the new variable `rows_imported` and give it the value 0 to start.

```
# create a variable for the number of rows imported
rows_imported = 0
for data in pd.read_csv(f'{file_dir}ratings.csv', chunksize=1000000):

    # print out the range of rows that are being imported

    data.to_sql(name='ratings', con=engine, if_exists='append')

    # increment the number of rows imported by the size of 'data'

    # print that the rows have finished importing
```

print out the range of rows that are being imported

When printing out monitoring information, it's generally a good practice to print out when a process is beginning and when a process has ended successfully, because if there's any problem, we have a better sense of which process caused the problem by seeing what part never finished successfully.

On top of this, it's good practice to keep both outputs on the same line, because it's easier to monitor which step is currently being performed. To

do this, we use the `end=` parameter in the `print` function. Setting the end to an empty string will prevent the output from going to the next line.

```
# create a variable for the number of rows imported
rows_imported = 0
for data in pd.read_csv(f'{file_dir}ratings.csv', chunksize=1000000):

    # print out the range of rows that are being imported
    print(f'importing rows {rows_imported} to {rows_imported + len(data)}...

    data.to_sql(name='ratings', con=engine, if_exists='append')

    # increment the number of rows imported by the size of 'data'

    # print that the rows have finished importing
```

increment the number of rows imported by the size of 'data'

This is a great time to use the compound operator `+=` to add the length of the data read in to `rows_imported`.

REWIND

Remember, compound operators are shortcuts to perform a simple arithmetic operation on a variable and reassign the new value to the variable.

For example, `foo += 1` is equivalent to `foo = foo + 1`.

```
# create a variable for the number of rows imported
rows_imported = 0
for data in pd.read_csv(f'{file_dir}ratings.csv', chunksize=1000000):
```

```
    # print out the range of rows that are being imported
    print(f'importing rows {rows_imported} to {rows_imported + len(data)}...

    data.to_sql(name='ratings', con=engine, if_exists='append')

    # increment the number of rows imported by the size of 'data'
    rows_imported += len(data)

    # print that the rows have finished importing
```

print that the rows have finished importing

Finally, we can print that everything was imported successfully. We don't need to specify an `end=` parameter in the `print` function since we do want a new line printed now.

```
# create a variable for the number of rows imported
rows_imported = 0
for data in pd.read_csv(f'{file_dir}ratings.csv', chunksize=1000000):

    # print out the range of rows that are being imported
    print(f'importing rows {rows_imported} to {rows_imported + len(data)}...

    data.to_sql(name='ratings', con=engine, if_exists='append')

    # increment the number of rows imported by the size of 'data'
    rows_imported += len(data)

    # print that the rows have finished importing
    print('Done.')
```

Now that we're done refactoring, we can delete our comments.

```
rows_imported = 0
for data in pd.read_csv(f'{file_dir}ratings.csv', chunksize=1000000):

    print(f'importing rows {rows_imported} to {rows_imported + len(data)}...
```

```
    data.to_sql(name='ratings', con=engine, if_exists='append')
    rows_imported += len(data)

    print(f'Done.')
```

Step 2: Print Elapsed Time

This is an optional step, but it's a good idea when running a long process. We're going to print the total amount of time elapsed at every step. This is useful to estimate how long the process is going to take.

We'll use the built-in `time` module in Python. `time.time()` returns the current time whenever it is called. Subtracting two time values gives the difference in seconds. By setting a variable at the beginning to the time at the start, inside the loop we can easily calculate elapsed time and print it out.

First, we'll add the following to our import cell and rerun it:

```
import time
```

Add two new comments: one before the `for` loop, and one inside the `for` loop, right before the last `final print()` statement. The first comment is to get the start time from `time.time()`, and the second comment is to add the elapsed time to the final printout.

```
rows_imported = 0
# get the start_time from time.time()
for data in pd.read_csv(f'{file_dir}ratings.csv', chunksize=1000000):

    print(f'importing rows {rows_imported} to {rows_imported + len(data)}...
    data.to_sql(name='ratings', con=engine, if_exists='append')
    rows_imported += len(data)

    # add elapsed time to final print out
    print(f'Done.')
```

get the start_time from time.time()

The `start_time = time.time()` method will initialize the `start_time` with the current time.

```
rows_imported = 0
# get the start_time from time.time()
start_time = time.time()
for data in pd.read_csv(f'{file_dir}ratings.csv', chunksize=1000000):
    print(f'importing rows {rows_imported} to {rows_imported + len(data)}...
    data.to_sql(name='ratings', con=engine, if_exists='append')
    rows_imported += len(data)

    # add elapsed time to final print out
    print(f'Done.')
```

The elapsed time is simply `time.time() - start_time`, which can be added directly into the f-string.

```
rows_imported = 0
# get the start_time from time.time()
start_time = time.time()
for data in pd.read_csv(f'{file_dir}ratings.csv', chunksize=1000000):
    print(f'importing rows {rows_imported} to {rows_imported + len(data)}...
    data.to_sql(name='ratings', con=engine, if_exists='append')
    rows_imported += len(data)

    # add elapsed time to final print out
    print(f'Done. {time.time() - start_time} total seconds elapsed')
```

Now we're ready to run this block of code and load the rating data into PostgreSQL. It may take some time to load, so you might want to use this time as a coffee break.

ADD/COMMIT/PUSH

Remember to add, commit, and push your work!

Once the cell finishes running, confirm the table imported correctly using pgAdmin. Verify the columns have the correct data type, inspect the first 100 rows, and check the row count.

If everything looks good, you are done. You just extracted really messy and almost unusable data, combed through it carefully to transform it, and then loaded it into a SQL database. Now the hackathon has a reliable, clean dataset just begging to be analyzed. Britta will definitely appreciate the hard work you put in.

NOTE

> Congrats on performing your first ETL process. By the way, ETL isn't the only way to create a data pipeline (even though it's the most common). There is also the **Extract, Load, and Transform (ELT)** paradigm.
>
> With ELT, data is stored as unstructured data in a data lake and transformed when analyses are performed. This requires very powerful analytical tools to perform the transformation tasks quickly, where ETL frontloads the transformation to make analyses easier to perform.