lab10

July 25, 2022

[1]:
```
# Initialize Otter
import otter
grader = otter.Notebook("lab10.ipynb")
```

1 Lab 10: SQL

In this lab, we are going to practice viewing, sorting, grouping, and merging tables with SQL. We will explore two datasets: 1. A "minified" version of the Internet Movie Database (IMDb). This SQLite database (~10MB) is a tiny sample of the much larger database (more than a few GBs). As a result, disclaimer that we may get wildly different results than if we use the whole database!

1. The money donated during the 2016 election using the Federal Election Commission (FEC)'s public records. You will be connecting to a SQLite database containing the data. The data we will be working with in this lab is quite small (~16MB); however, it is a sample taken from a much larger database (more than a few GBs).

Due Date: Saturday, July 30, 11:59 PM PT.

1.0.1 Collaboration Policy

Data science is a collaborative activity. While you may talk with others about this assignment, we ask that you **write your solutions individually**. If you discuss the assignment with others, please **include their names** in the cell below.

Collaborators: *list names here*

[2]:
```
# Run this cell to set up your notebook
import numpy as np
import pandas as pd
import plotly.express as px
import sqlalchemy
from ds100_utils import fetch_and_cache
from pathlib import Path
%load_ext sql

# Unzip the data.
!unzip -o data.zip
```

```
Archive:  data.zip
  inflating: imdbmini.db
  inflating: fec_nyc.db
```

1.1 SQL Query Syntax

Throughout this lab, you will become familiar with the following syntax for the `SELECT` query:

```
SELECT [DISTINCT]
    {* | expr [[AS] c_alias]
    {,expr [[AS] c_alias] ...}}
FROM tableref {, tableref}
[[INNER | LEFT ] JOIN table_name
    ON qualification_list]
[WHERE search_condition]
[GROUP BY colname {,colname...}]
[HAVING search condition]
[ORDER BY column_list]
[LIMIT number]
[OFFSET number of rows];
```

2 Part 0 [Tutorial]: Writing SQL in Jupyter Notebooks

2.1 1. %%sql cell magic

In lecture, we used the `sql` extension to call **%%sql cell magic**, which enables us to connect to SQL databses and issue SQL commands within Jupyter Notebooks.

Run the below cell to connect to a mini IMDb database.

[3]:
```
%sql sqlite:///imdbmini.db
```

Above, prefixing our single-line command with `%sql` means that the entire line will be treated as a SQL command (this is called "line magic"). In this class we will most often write multi-line SQL, meaning we need "cell magic", where the first line has `%%sql` (note the double `%` operator).

The database `imdbmini.db` includes several tables, one of which is `Title`. Running the below cell will return first 5 lines of that table. Note that `%%sql` is on its own line.

We've also included syntax for single-line comments, which are surrounded by `--`.

[4]:
```
%%sql
/*
 * This is a
 * multi-line comment.
 */
-- This is a single-line/inline comment. --
SELECT *
FROM Name
LIMIT 5;
```

```
 * sqlite:///imdbmini.db
Done.
```

[4]:
```
[(1, 'Fred Astaire', '1899', '1987', 'soundtrack,actor,miscellaneous'),
 (2, 'Lauren Bacall', '1924', '2014', 'actress,soundtrack'),
 (3, 'Brigitte Bardot', '1934', None, 'actress,soundtrack,music_department'),
 (4, 'John Belushi', '1949', '1982', 'actor,soundtrack,writer'),
 (5, 'Ingmar Bergman', '1918', '2007', 'writer,director,actor')]
```

2.1.1 2. The Pandas command `pd.read_sql`

As of 2022, the `%sql` magic for Jupyter Notebooks is still in development (check out its GitHub. It is still missing many features that would justify real-world use with Python. In particular, its returned tables are *not* Pandas dataframes (for example, the query result from the above cell is missing an index).

The rest of this section describes how data scientists use SQL and Python in practice, using the Pandas command `pd.read_sql` (documentation). **You will see both `%sql` magic and `pd.read_sql` in this course**.

The below cell connects to the same database using the SQLAlchemy Python library, which can connect to several different database management systems, including sqlite3, MySQL, PostgreSQL, and Oracle. The library also supports an advanced feature for generating queries called an object relational mapper or ORM, which we won't discuss in this course but is quite useful for application development.

[5]:
```
# important!!! run this cell
import sqlalchemy

# create a SQL Alchemy connection to the database
engine = sqlalchemy.create_engine("sqlite:///imdbmini.db")
connection = engine.connect()
```

With the SQLAlchemy object `connection`, we can then call `pd.read_sql` which takes in a **`query` string**. Note the `"""` to define our multi-line string, which allows us to have a query span multiple lines. The resulting `df` DataFrame stores the results of the same SQL query from the previous section.

[6]:
```
# just run this cell
query = """
SELECT *
FROM Title
LIMIT 5;
"""

df = pd.read_sql(query, engine)
df
```

[6]:
```
   tconst titleType                primaryTitle  \
0     417     short           A Trip to the Moon
1    4972     movie        The Birth of a Nation
2   10323     movie  The Cabinet of Dr. Caligari
3   12349     movie                      The Kid
4   13442     movie                    Nosferatu

                          originalTitle isAdult startYear endYear  \
0                 Le voyage dans la lune       0      1902    None
1                  The Birth of a Nation       0      1915    None
2           Das Cabinet des Dr. Caligari       0      1920    None
3                                The Kid       0      1921    None
4  Nosferatu, eine Symphonie des Grauens       0      1922    None

  runtimeMinutes                    genres
0             13  Action,Adventure,Comedy
1            195         Drama,History,War
2             76   Fantasy,Horror,Mystery
3             68      Comedy,Drama,Family
4             94           Fantasy,Horror
```

Long error messages: Given that the SQL query is now in the string, the errors become more unintelligible. Consider the below (incorrect) query, which has a semicolon in the wrong place.

[7]:
```
# uncomment the below code and check out the error

# query = """
# SELECT *
# FROM Title;
# LIMIT 5
# """
# pd.read_sql(query, engine)
```

Now that's an unruly error message!

2.1.2 3. A suggested workflow for writing SQL in Jupyter Notebooks

Which approach is better, `%sql` magic or `pd.read_sql`?

The SQL database generally contains much more data than what you would analyze in detail. As a Python-fluent data scientist, you will often query SQL databases to perform initial exploratory data analysis, a subset of which you load into Python for further processing.

In practice, you would likely use a combination of the two approaches. First, you'd try out some SQL queries with `%sql` magic to get an interesting subset of data. Then, you'd copy over the query into a `pd.read_sql` command for visualization, modeling, and export with Pandas, sklearn, and other Python libraries.

For SQL assignments in this course, to minimize unruly error messages while maximizing Python compatibility, we suggest the following "sandboxed" workflow: 1. Create a `%%sql` magic cell **below**

the answer cell. You can copy in the below code:

```
%% sql
-- This is a comment. Put your code here... --
```

1. Work on the SQL query in the `%%sql` cell; e.g., `SELECT ... ;`
2. Then, once you're satisfied with your SQL query, copy it into the multi-string query in the answer cell (the one that contains the `pd.read_sql` call).

You don't have to follow the above workflow to get full credit on assignments, but we suggest it to reduce debugging headaches. We've created the scratchwork `%%sql` cells for you in this assignment, but **do not** add cells between this `%%sql` cell and the Python cell right below it. It will cause errors when we run the autograder, and it will sometimes cause a failure to generate the PDF file.

3 Part 1: The IMDb (mini) Dataset

Let's explore a miniature version of the IMDb Dataset. This is the same dataset that we will use for the upcoming homework.

Let's load in the database in two ways (using both Python and cell magic) so that we can flexibly explore the database.

[8]:
```
engine = sqlalchemy.create_engine("sqlite:///imdbmini.db")
connection = engine.connect()
```

[9]:
```
%sql sqlite:///imdbmini.db
```

[10]:
```
%%sql
SELECT * FROM sqlite_master WHERE type='table';
```

```
 * sqlite:///imdbmini.db
Done.
```

[10]:
```
[('table', 'Title', 'Title', 2, 'CREATE TABLE "Title" (\n"tconst" INTEGER,\n
"titleType" TEXT,\n  "primaryTitle" TEXT,\n  "originalTitle" TEXT,\n  "isAdult"
TEXT,\n  "startYear" TEXT,\n  "endYear" TEXT,\n  "runtimeMinutes" TEXT,\n
"genres" TEXT\n)'),
 ('table', 'Name', 'Name', 12, 'CREATE TABLE "Name" (\n"nconst" INTEGER,\n
"primaryName" TEXT,\n  "birthYear" TEXT,\n  "deathYear" TEXT,\n
"primaryProfession" TEXT\n)'),
 ('table', 'Role', 'Role', 70, 'CREATE TABLE "Role" (\ntconst INTEGER,\nordering
TEXT,\nnconst INTEGER,\ncategory TEXT,\njob TEXT,\ncharacters TEXT\n)'),
 ('table', 'Rating', 'Rating', 41, 'CREATE TABLE "Rating" (\ntconst
INTEGER,\naverageRating TEXT,\nnumVotes TEXT\n)')]
```

From running the above cell, we see the database has 4 tables: `Name`, `Role`, `Rating`, and `Title`.

[Click to Expand] See descriptions of each table's schema.

`Name` – Contains the following information for names of people.

- nconst (text) - alphanumeric unique identifier of the name/person
- primaryName (text)– name by which the person is most often credited
- birthYear (integer) – in YYYY format
- deathYear (integer) – in YYYY format

`Role` – Contains the principal cast/crew for titles.

- tconst (text) - alphanumeric unique identifier of the title
- ordering (integer) – a number to uniquely identify rows for a given tconst
- nconst (text) - alphanumeric unique identifier of the name/person
- category (text) - the category of job that person was in
- characters (text) - the name of the character played if applicable, else '\N'

`Rating` – Contains the IMDb rating and votes information for titles.

- tconst (text) - alphanumeric unique identifier of the title
- averageRating (real) – weighted average of all the individual user ratings
- numVotes (integer) - number of votes (i.e., ratings) the title has received

`Title` - Contains the following information for titles.

- tconst (text) - alphanumeric unique identifier of the title
- titleType (text) - the type/format of the title
- primaryTitle (text) - the more popular title / the title used by the filmmakers on promotional materials at the point of release
- isAdult (text) - 0: non-adult title; 1: adult title
- year (YYYY) – represents the release year of a title.
- runtimeMinutes (integer) – primary runtime of the title, in minutes

From the above descriptions, we can conclude the following: * `Name.nconst` and `Title.tconst` are primary keys of the `Name` and `Title` tables, respectively. * that `Role.nconst` and `Role.tconst` are **foreign keys** that point to `Name.nconst` and `Title.tconst`, respectively.

3.1 Question 1

What are the different kinds of `titleTypes` included in the `Title` table? Write a query to find out all the unique `titleTypes` of films using the `DISTINCT` keyword. (**You may not use `GROUP BY`.**)

[11]:
```
%%sql
/*
 * Code in this scratchwork cell is __not graded.__
 * Copy over any SQL queries you write here into the below Python cell.
 * Do __not__ insert any new cells in between the SQL/Python cells!
 * Doing so may break the autograder.
 */
-- Write below this comment. --
```

```
 * sqlite:///imdbmini.db
0 rows affected.
```

[11]:
```
[]
```

[12]:
```
query_q1 = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q1 = """
SELECT
    DISTINCT titleType
FROM Title;
"""
# END SOLUTION
res_q1 = pd.read_sql(query_q1, engine)
res_q1
```

[12]:

	titleType
0	short
1	movie
2	tvSeries
3	tvMovie
4	tvMiniSeries
5	video
6	videoGame
7	tvEpisode
8	tvSpecial

[13]:
```
grader.check("q1")
```

[13]:
```
q1 results: All test cases passed!
```

3.2 Question 2

Before we proceed we want to get a better picture of the kinds of jobs that exist. To do this examine the `Role` table by computing the number of records with each job `category`. Present the results in descending order by the total counts.

The top of your table should look like this (however, you should have more rows):

	category	total
0	actor	21665
1	writer	13830
2	...	...

[14]:
```
%%sql
/*
 * Code in this scratchwork cell is __not graded.__
 * Copy over any SQL queries you write here into the below Python cell.
 * Do __not__ insert any new cells in between the SQL/Python cells!
 * Doing so may break the autograder.
 */
-- Write below this comment. --
```

```
 * sqlite:///imdbmini.db
0 rows affected.
```

[14]:
```
[]
```

[15]:
```
query_q2 = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q2 = """
SELECT
    category,
    Count(*) AS total
FROM Role
GROUP BY category
ORDER BY total DESC;
"""
# END SOLUTION
res_q2 = pd.read_sql(query_q2, engine)
res_q2
```

[15]:

	category	total
0	actor	21665
1	writer	13830
2	actress	12175
3	producer	11028
4	director	6995
5	composer	4123
6	cinematographer	2747
7	editor	1558
8	self	623
9	production_designer	410
10	archive_footage	66
11	archive_sound	6

[16]:
```
grader.check("q2")
```

[16]:
```
q2 results: All test cases passed!
```

If we computed the results correctly we should see a nice horizontal bar chart of the counts per category below:

3.3 Question 3

Now that we have a better sense of the basics of our data, we can ask some more interesting questions.

The `Rating` table has the `numVotes` and the `averageRating` for each title. Which 10 films have the most ratings?

Write a SQL query that outputs three fields: the `title`, `numVotes`, and `averageRating` for the 10 films that have the highest number of ratings. Sort the result in descending order by the number of votes.

Hint: The `numVotes` in the `Rating` table is not an integer! Use `CAST(Rating.numVotes AS int) AS numVotes` to convert the attribute to an integer.

[17]:
```
%%sql
/*
 * Code in this scratchwork cell is __not graded.__
 * Copy over any SQL queries you write here into the below Python cell.
 * Do __not__ insert any new cells in between the SQL/Python cells!
 * Doing so may break the autograder.
 */
-- Write below this comment. --
```

```
 * sqlite:///imdbmini.db
0 rows affected.
```

[17]:
```
[]
```

[18]:
```
query_q3 = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q3 = """
SELECT
    Title.primaryTitle AS title,
    CAST(Rating.numVotes AS int) AS numVotes,
    Rating.averageRating as averageRating
FROM Title
JOIN Rating
    ON Rating.tconst = Title.tconst
ORDER BY numVotes DESC
```

```
LIMIT 10;
"""
# END SOLUTION

res_q3 = pd.read_sql(query_q3, engine)
res_q3
```

[18]:

	title	numVotes	averageRating
0	The Shawshank Redemption	2462686	9.3
1	The Dark Knight	2417875	9.0
2	Inception	2169255	8.8
3	Fight Club	1939312	8.8
4	Pulp Fiction	1907561	8.9
5	Forrest Gump	1903969	8.8
6	Game of Thrones	1874040	9.2
7	The Matrix	1756469	8.7
8	The Lord of the Rings: The Fellowship of the Ring	1730296	8.8
9	The Lord of the Rings: The Return of the King	1709023	8.9

[19]:
```
grader.check("q3")
```

[19]:
```
q3 results: All test cases passed!
```

4 Part 2: Election Donations in New York City

Finally, let's analyze the Federal Election Commission (FEC)'s public records. We connect to the database in two ways (using both Python and cell magic) so that we can flexibly explore the database.

[20]:
```
# important!!! run this cell and the next one
import sqlalchemy
# create a SQL Alchemy connection to the database
engine = sqlalchemy.create_engine("sqlite:///fec_nyc.db")
connection = engine.connect()
```

[21]:
```
%sql sqlite:///fec_nyc.db
```

4.1 Table Descriptions

Run the below cell to explore the **schemas** of all tables saved in the database.

If you'd like, you can consult the below linked FEC pages for the descriptions of the tables themselves.

- `cand` (link): Candidates table. Contains names and party affiliation.
- `comm` (link): Committees table. Contains committee names and types.
- `indiv_sample_nyc` (link): All individual contributions from New York City .

[22]:
```
%%sql
/* just run this cell */
SELECT sql FROM sqlite_master WHERE type='table';
```

```
 * sqlite:///fec_nyc.db
   sqlite:///imdbmini.db
Done.
```

[22]:
```
[('CREATE TABLE "cand" (\n    cand_id character varying(9),\n    cand_name
text,\n    cand_pty_affiliation character varying(3),\n    cand_election_yr i
… (196 characters truncated) … er varying(9),\n    cand_st1 text,\n
cand_st2 text,\n    cand_city text,\n    cand_st character varying(2),\n
cand_zip character varying(10)\n)',),
 ('CREATE TABLE "comm"(\n  "cmte_id" TEXT,\n  "cmte_nm" TEXT,\n  "tres_nm"
TEXT,\n  "cmte_st1" TEXT,\n  "cmte_st2" TEXT,\n  "cmte_city" TEXT,\n  "cmte_s
… (46 characters truncated) … XT,\n  "cmte_tp" TEXT,\n
"cmte_pty_affiliation" TEXT,\n  "cmte_filing_freq" TEXT,\n  "org_tp" TEXT,\n
"connected_org_nm" TEXT,\n  "cand_id" TEXT\n)',),
 ('CREATE TABLE indiv_sample_nyc (\n    cmte_id character varying(9),\n
amndt_ind character(1),\n    rpt_tp character varying(3),\n    transaction_pg
… (299 characters truncated) …   transaction_amt integer,\n    other_id
text,\n    tran_id text,\n    file_num bigint,\n    memo_cd text,\n    memo_text
text,\n    sub_id bigint\n)',)]
```

Let's look at the `indiv_sample_nyc` table. The below cell displays individual donations made by residents of the state of New York. We use `LIMIT 5` to avoid loading and displaying a huge table.

[23]:
```
%%sql
/* just run this cell */
SELECT *
FROM indiv_sample_nyc
LIMIT 5;
```

```
 * sqlite:///fec_nyc.db
   sqlite:///imdbmini.db
Done.
```

[23]:
```
[('C00445015', 'N', 'Q1', 'P    ', 15951128130, '15', 'IND', 'SINGER, TRIPP
MR.', 'NEW YORK', 'NY', '100214505', 'ATLANTIC MAILBOXES, INC.', 'OWNER',
'01302015', 1000, '', 'A-CF13736', 1002485, '', '', 4041420151241812398),
 ('C00510461', 'N', 'Q1', 'P    ', 15951129284, '15E', 'IND', 'SIMON, DANIEL A',
'NEW YORK', 'NY', '100237940', 'N/A', 'RETIRED', '03292015', 400, 'C00401224',
'VN8JBDDJBA8', 1002590, '', '* EARMARKED CONTRIBUTION: SEE BELOW',
4041420151241813640),
 ('C00422410', 'N', 'Q1', 'P    ', 15970352211, '15', 'IND', 'ABDUL RAUF,
FEISAL', 'NEW YORK', 'NY', '101150010', 'THE CORDOBA INITIATIVE', 'CHAIRMAN',
'03042015', 250, '', 'VN8A3DBSYG6', 1003643, '', '', 4041620151241914560),
 ('C00510461', 'N', 'Q1', 'P    ', 15951129280, '15', 'IND', 'SCHWARZER, FRANK',
```

```
'NEW YORK', 'NY', '100145135', 'METRO HYDRAULIC JACK CO', 'SALES', '01162015',
100, '', 'VN8JBDAP4C4', 1002590, '', '* EARMARKED CONTRIBUTION: SEE BELOW',
4041420151241813630),
 ('C00510461', 'N', 'Q1', 'P    ', 15951129281, '15', 'IND', 'SCHWARZER, FRANK',
'NEW YORK', 'NY', '100145135', 'METRO HYDRAULIC JACK CO', 'SALES', '02162015',
100, '', 'VN8JBDBRDG3', 1002590, '', '* EARMARKED CONTRIBUTION: SEE BELOW',
4041420151241813632)]
```

You can write a SQL query to return the id and name of the first five candidates from the Democratic party, as below:

[24]:
```
%%sql
/* just run this cell */
SELECT cand_id, cand_name
FROM cand
WHERE cand_pty_affiliation = 'DEM'
LIMIT 5;
```

```
 * sqlite:///fec_nyc.db
   sqlite:///imdbmini.db
Done.
```

[24]:
```
[('H0AL05049', 'CRAMER, ROBERT E "BUD" JR'),
 ('H0AL07086', 'SEWELL, TERRYCINA ANDREA'),
 ('H0AL07094', 'HILLIARD, EARL FREDERICK JR'),
 ('H0AR01091', 'GREGORY, JAMES CHRISTOPHER'),
 ('H0AR01109', 'CAUSEY, CHAD')]
```

4.2 [Tutorial] Matching Text with `LIKE`

First, let's look at 2016 election contributions made by Donald Trump, who was a New York (NY) resident during that year. The following SQL query returns the `cmte_id`, `transaction_amt`, and `name` for every contribution made by any donor with "DONALD" and "TRUMP" in their name in the `indiv_sample_nyc` table.

Notes: * We use the `WHERE ... LIKE '...'` to match fields with text patterns. The `%` wildcard represents at least zero characters. Compare this to what you know from regex! * We use `pd.read_sql` syntax here because we will do some EDA on the result `res`.

[25]:
```
# just run this cell
example_query = """
SELECT
    cmte_id,
    transaction_amt,
    name
FROM indiv_sample_nyc
WHERE name LIKE '%TRUMP%' AND name LIKE '%DONALD%';
"""
```

```
example_res = pd.read_sql(example_query, engine)
example_res
```

[25]:

	cmte_id	transaction_amt	name
0	C00230482	2600	DONALD, TRUMP
1	C00230482	2600	DONALD, TRUMP
2	C00014498	9000	TRUMP, DONALD
3	C00494229	2000	TRUMP, DONALD MR
4	C00571869	2700	TRUMP, DONALD J.
..	…	…	…
152	C00608489	5	DONALD J TRUMP FOR PRESIDENT INC
153	C00608489	5	DONALD J TRUMP FOR PRESIDENT INC
154	C00608489	5	DONALD J TRUMP FOR PRESIDENT INC
155	C00608489	5	DONALD J TRUMP FOR PRESIDENT INC
156	C00608489	5	DONALD J TRUMP FOR PRESIDENT INC

[157 rows x 3 columns]

If we look at the list above, it appears that some donations were not by Donald Trump himself, but instead by an entity called "DONALD J TRUMP FOR PRESIDENT INC". Fortunately, we see that our query only seems to have picked up one such anomalous name.

[26]:
```
# just run this cell
example_res['name'].value_counts()
```

[26]:
```
TRUMP, DONALD J.                    133
DONALD J TRUMP FOR PRESIDENT INC     15
TRUMP, DONALD                         4
DONALD, TRUMP                         2
TRUMP, DONALD MR                      1
TRUMP, DONALD J MR.                   1
TRUMP, DONALD J MR                    1
Name: name, dtype: int64
```

4.3 Question 4

Revise the above query so that the 15 anomalous donations made by "DONALD J TRUMP FOR PRESIDENT INC" do not appear. Your resulting table should have 142 rows.

Hints: * Consider using the above query as a starting point, or checking out the SQL query skeleton at the top of this lab. * The `NOT` keyword may also be useful here.

[27]:
```
%%sql
/*
 * Code in this scratchwork cell is __not graded.__
 * Copy over any SQL queries you write here into the below Python cell.
 * Do __not__ insert any new cells in between the SQL/Python cells!
```

```
 * Doing so may break the autograder.
 */
-- Write below this comment. --
```

```
 * sqlite:///fec_nyc.db
   sqlite:///imdbmini.db
0 rows affected.
```

[27]:
```
[]
```

[28]:
```
query_q4 = """
...       # replace this with
...;      # your multi-line solution
"""
# BEGIN SOLUTION NO PROMPT
query_q4 = """
SELECT
    cmte_id,
    transaction_amt,
    name
FROM indiv_sample_nyc
WHERE name LIKE '%TRUMP%'
    AND name LIKE '%DONALD%'
    AND name NOT LIKE '%INC%';
"""
# END SOLUTION

res_q4 = pd.read_sql(query_q4, engine)
res_q4
```

[28]:

	cmte_id	transaction_amt	name
0	C00230482	2600	DONALD, TRUMP
1	C00230482	2600	DONALD, TRUMP
2	C00014498	9000	TRUMP, DONALD
3	C00494229	2000	TRUMP, DONALD MR
4	C00571869	2700	TRUMP, DONALD J.
..	…	…	…
137	C00580100	9752	TRUMP, DONALD J.
138	C00580100	2574	TRUMP, DONALD J.
139	C00580100	23775	TRUMP, DONALD J.
140	C00580100	2000000	TRUMP, DONALD J.
141	C00580100	2574	TRUMP, DONALD J.

[142 rows x 3 columns]

[29]:
```
grader.check("q4")
```

[29]:
```
q4 results: All test cases passed!
```

4.4 Question 5: `JOIN`ing Tables

Let's explore the other two tables in our database: `cand` and `comm`.

The `cand` table contains summary financial information about each candidate registered with the FEC or appearing on an official state ballot for House, Senate or President.

[30]:
```
%%sql
/* just run this cell */
SELECT *
FROM cand
LIMIT 5;
```

```
 * sqlite:///fec_nyc.db
   sqlite:///imdbmini.db
Done.
```

[30]:
```
[('H0AK00097', 'COX, JOHN R.', 'REP', 2014, 'AK', 'H', 0, 'C', 'N', 'C00525261',
'P.O. BOX 1092 ', '', 'ANCHOR POINT', 'AK', '99556'),
 ('H0AL02087', 'ROBY, MARTHA', 'REP', 2016, 'AL', 'H', 2, 'I', 'C', 'C00462143',
'PO BOX 195', '', 'MONTGOMERY', 'AL', '36101'),
 ('H0AL02095', 'JOHN, ROBERT E JR', 'IND', 2016, 'AL', 'H', 2, 'C', 'N', '',
'1465 W OVERBROOK RD', '', 'MILLBROOK', 'AL', '36054'),
 ('H0AL05049', 'CRAMER, ROBERT E "BUD" JR', 'DEM', 2008, 'AL', 'H', 5, '', 'P',
'C00239038', 'PO BOX 2621', '', 'HUNTSVILLE', 'AL', '35804'),
 ('H0AL05163', 'BROOKS, MO', 'REP', 2016, 'AL', 'H', 5, 'I', 'C', 'C00464149',
'7610 FOXFIRE DRIVE', '', 'HUNTSVILLE', 'AL', '35802')]
```

The `comm` table contains summary financial information about each committee registered with the FEC. Committees are organizations that spend money for political action or parties, or spend money for or against political candidates.

[31]:
```
%%sql
/* just run this cell */
SELECT *
FROM comm
LIMIT 5;
```

```
 * sqlite:///fec_nyc.db
   sqlite:///imdbmini.db
Done.
```

[31]:
```
[('C00000059', 'HALLMARK CARDS PAC', 'ERIN BROWER', '2501 MCGEE', 'MD#288',
'KANSAS CITY', 'MO', '64108', 'U', 'Q', 'UNK', 'M', 'C', '', ''),
 ('C00000422', 'AMERICAN MEDICAL ASSOCIATION POLITICAL ACTION COMMITTEE',
'WALKER, KEVIN', '25 MASSACHUSETTS AVE, NW', 'SUITE 600', 'WASHINGTON', 'DC',
'20001', 'B', 'Q', '', 'M', 'M', 'AMERICAN MEDICAL ASSOCIATION', ''),
```

```
 ('C00000489', 'D R I V E POLITICAL FUND CHAPTER 886', 'TOM RITTER', '3528 W
RENO', '', 'OKLAHOMA CITY', 'OK', '73107', 'U', 'N', '', 'Q', 'L', 'TEAMSTERS
LOCAL UNION 886', ''),
 ('C00000547', 'KANSAS MEDICAL SOCIETY POLITICAL ACTION COMMITTEE', 'C. RICHARD
BONEBRAKE, M.D.', '623 SW 10TH AVE', '', 'TOPEKA', 'KS', '66612', 'U', 'Q',
'UNK', 'Q', 'T', '', ''),
 ('C00000638', 'INDIANA STATE MEDICAL ASSOCIATION POLITICAL ACTION COMMITTEE',
'VIDYA KORA, M.D.', '322 CANAL WALK, CANAL LEVEL', '', 'INDIANAPOLIS', 'IN',
'46202', 'U', 'Q', '', 'Q', 'M', '', '')]
```

4.4.1 Question 5a

Notice that both the `cand` and `comm` tables have a `cand_id` column. Let's try joining these two tables on this column to print out committee information for candidates.

List the first 5 candidate names (`cand_name`) in reverse lexicographic order by `cand_name`, along with their corresponding committee names. **Only select rows that have a matching `cand_id` in both tables.**

Your output should look similar to the following:

	cand_name	cmte_nm
0	ZUTLER, DANIEL PAUL MR	CITIZENS TO ELECT DANIEL P ZUTLER FOR PRESIDENT
1	ZUMWALT, JAMES	ZUMWALT FOR CONGRESS
...	...	...

Consider starting from the following query skeleton, which uses the `AS` keyword to rename the `cand` and `comm` tables to `c1` and `c2`, respectively. Which join is most appropriate?

```
SELECT ...
FROM cand AS c1
    [INNER | {LEFT |RIGHT | FULL } {OUTER}] JOIN comm AS c2
    ON ...
...
...;
```

[32]:
```
%%sql
/*
 * Code in this scratchwork cell is __not graded.__
 * Copy over any SQL queries you write here into the below Python cell.
 * Do __not__ insert any new cells in between the SQL/Python cells!
 * Doing so may break the autograder.
 */
-- Write below this comment. --
```

```
 * sqlite:///fec_nyc.db
   sqlite:///imdbmini.db
0 rows affected.
```

[32]:
```
[]
```

[33]:
```
query_q5a = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q5a = """
SELECT c1.cand_name, c2.cmte_nm
FROM cand AS c1
JOIN comm AS c2   -- INNER JOIN --
    ON c1.cand_id = c2.cand_id
ORDER BY c1.cand_name DESC
LIMIT 5
"""
# END SOLUTION
res_q5a = pd.read_sql(query_q5a, engine)
res_q5a
```

[33]:

	cand_name	cmte_nm
0	ZUTLER, DANIEL PAUL MR	CITIZENS TO ELECT DANIEL P ZUTLER FOR PRESIDENT
1	ZUMWALT, JAMES	ZUMWALT FOR CONGRESS
2	ZUKOWSKI, ANDREW GEORGE	ZUKOWSKI FOR CONGRESS
3	ZUCCOLO, JOE	JOE ZUCCOLO FOR CONGRESS
4	ZORN, ROBERT ERWIN	CONSTITUTIONAL COMMITTEE

[34]:
```
grader.check("q5a")
```

[34]:
```
q5a results: All test cases passed!
```

4.4.2 Question 5b

Suppose we modify the query from the previous part to include *all* candidates, **including those that don't have a committee.**

List the first 5 candidate names (`cand_name`) in reverse lexicographic order by `cand_name`, along with their corresponding committee names. If the candidate has no committee in the `comm` table, then `cmte_nm` should be NULL (or None in the Python representation).

Your output should look similar to the following:

	cand_name	cmte_nm
0	ZUTLER, DANIEL PAUL MR	CITIZENS TO ELECT DANIEL P ZUTLER FOR PRESIDENT
...	...	...
4	ZORNOW, TODD MR	None

Hint: Start from the same query skeleton as the previous part. Which join is most appropriate?

[35]:
```
%%sql
/*
 * Code in this scratchwork cell is __not graded.__
 * Copy over any SQL queries you write here into the below Python cell.
 * Do __not__ insert any new cells in between the SQL/Python cells!
 * Doing so may break the autograder.
 */
-- Write below this comment. --
```

```
 * sqlite:///fec_nyc.db
   sqlite:///imdbmini.db
0 rows affected.
```

[35]:
```
[]
```

[36]:
```
query_q5b = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q5b = """
SELECT c1.cand_name, c2.cmte_nm
FROM cand AS c1
LEFT JOIN comm AS c2 -- LEFT OUTER JOIN --
    ON c1.cand_id = c2.cand_id
ORDER BY c1.cand_name DESC
LIMIT 5;
"""
# END SOLUTION
res_q5b = pd.read_sql(query_q5b, engine)
res_q5b
```

[36]:
```
                cand_name                                          cmte_nm
0   ZUTLER, DANIEL PAUL MR  CITIZENS TO ELECT DANIEL P ZUTLER FOR PRESIDENT
1           ZUMWALT, JAMES                              ZUMWALT FOR CONGRESS
2  ZUKOWSKI, ANDREW GEORGE                            ZUKOWSKI FOR CONGRESS
```

```
3             ZUCCOLO, JOE                          JOE ZUCCOLO FOR CONGRESS
4          ZORNOW, TODD MR                                              None
```

[37]:
```
grader.check("q5b")
```

[37]:
```
q5b results: All test cases passed!
```

4.5 Question 6: Subqueries and Grouping (OPTIONAL)

If we return to our results from Question 4, we see that many of the contributions were to the same committee:

[38]:
```
# Your SQL query result from Question 4
# reprinted for your convenience
res_q4['cmte_id'].value_counts()
```

[38]:
```
C00580100    131
C00230482      2
C00571869      2
C00014498      1
C00494229      1
C00136457      1
C00034033      1
C00554949      1
C00369033      1
C00055582      1
Name: cmte_id, dtype: int64
```

Create a new SQL query that returns the total amount that Donald Trump contributed to each committee.

Your table should have four columns: `cmte_id`, `total_amount` (total amount contributed to that committee), `num_donations` (total number of donations), and `cmte_nm` (name of the committee). Your table should be sorted in **decreasing order** of `total_amount`.

This is a hard question! Don't be afraid to reference the lecture slides, or the overall SQL query skeleton at the top of this lab.

Here are some other hints:

- Note that committee names are not available in `indiv_sample_nyc`, so you will have to obtain information somehow from the `comm` table (perhaps a `JOIN` would be useful).
- Remember that you can compute summary statistics after grouping by using aggregates like `COUNT(*)`, `SUM()` as output fields.
- A **subquery** may be useful to break your question down into subparts. Consider the following query skeleton, which uses the `WITH` operator to store a subquery's results in a temporary table named `donations`.

```
WITH donations AS (
    SELECT ...
    FROM ...
    ... JOIN ...
        ON ...
    WHERE ...
)
SELECT ...
FROM donations
GROUP BY ...
ORDER BY ...;
```

[39]:
```
%%sql
/*
 * Code in this scratchwork cell is __not graded.__
 * Copy over any SQL queries you write here into the below Python cell.
 * Do __not__ insert any new cells in between the SQL/Python cells!
 * Doing so may break the autograder.
 */
-- Write below this comment. --
```

```
 * sqlite:///fec_nyc.db
   sqlite:///imdbmini.db
0 rows affected.
```

[39]: []

[40]:
```
query_q6 = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q6 = """
WITH donations AS (
    SELECT
        comm.cmte_id,
        comm.cmte_nm,
        transaction_amt,
        name
    FROM indiv_sample_nyc
    JOIN comm
        ON indiv_sample_nyc.cmte_id == comm.cmte_id
    WHERE name LIKE '%TRUMP%' AND name LIKE '%DONALD%' AND name NOT LIKE '%INC'
)
SELECT
    cmte_id,
    SUM(transaction_amt) as total_amount,
```

```
    COUNT(*) as num_donations,
    cmte_nm
FROM donations
GROUP BY cmte_id
ORDER BY total_amount DESC
"""
# END SOLUTION

res_q6 = pd.read_sql(query_q6, engine)
res_q6
```

[40]:

	cmte_id	total_amount	num_donations	cmte_nm
0	C00580100	18633157	131	DONALD J. TRUMP FOR PRESIDENT, INC.
1	C00055582	10000	1	NY REPUBLICAN FEDERAL CAMPAIGN COMMITTEE
2	C00014498	9000	1	REPUBLICAN PARTY OF IOWA
3	C00571869	5400	2	DONOVAN FOR CONGRESS
4	C00230482	5200	2	GRASSLEY COMMITTEE INC
5	C00136457	5000	1	NEW HAMPSHIRE REPUBLICAN STATE COMMITTEE
6	C00034033	5000	1	SOUTH CAROLINA REPUBLICAN PARTY
7	C00554949	2600	1	FRIENDS OF DAVE BRAT INC.
8	C00494229	2000	1	HELLER FOR SENATE
9	C00369033	1000	1	TEXANS FOR SENATOR JOHN CORNYN INC

[41]:
```
grader.check("q6")
```

[41]:
```
q6 results: All test cases passed!
```

5 Congratulations! You finished the lab!

To double-check your work, the cell below will rerun all of the autograder tests.

[42]:
```
grader.check_all()
```

[42]:
```
q1 results: All test cases passed!

q2 results: All test cases passed!

q3 results: All test cases passed!

q4 results: All test cases passed!

q5a results: All test cases passed!

q5b results: All test cases passed!

q6 results: All test cases passed!
```

5.1 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

[43]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export(pdf=False)
```

```
<IPython.core.display.HTML object>
```