Lab Assignment 7: Database Queries

DS 6001

Instructions

Please answer the following questions as completely as possible using text, code, and the results of code as needed. Format your answers in a Jupyter notebook. To receive full credit, make sure you address every part of the problem, and make sure your document is formatted in a clean and professional way.

Jillian Howe | took my extension for this one bc i didn't finish in time but little did i know that my docker and conda would stop working . wasn't able to test any code after any started working with postgres. tried to fix it and broke conda on my computer. PARTS 2 + 3 I havent been able to run on my local system. I checked with other students to make sure I had similar code and undesrtood what was happening even if my computer wasn't working

I will resubmit with the right params and such once i get my conda/ docker working again but I am past my due date so here we are. Apoligize because i couldnt restart/rerun or check my own code :/

Problem 0

Import the following packages:

In [2]:
```
import numpy as np
import pandas as pd
import sys
import os
import requests
import psycopg
import pymongo
import json
from bson.json_util import dumps, loads
from sqlalchemy import create_engine
import dotenv
```

Problem 1

I think we teach a few subjects all wrong. The first class in statistics, for example, usually applies hypothesis testing to dumb and unrealistic examples, and involves looking-up p-values in the back of a heavy textbook. Of course this dissuades many potential statisticians from pursuing the subject. But if we instead placed the focus on probability, experimentation, and prediction and applied the methods to fun or interesting examples like games and current science, then we would make the subject much more appealing in its own right, and not just due to its associations with high income employment.

I think database work and SQL get a similarly bad treatment.

SQL is a universal language for interacting with relational databases. MySQL, SQlite, PostgreSQL, Oracle, Microsoft, and every other relational database management system uses SQL to connect to data. SQL can be run inside Python, R, Javascript, and many other interpretive programming languages. It is the single best language for collaboration. It is far older than most programming languages yet is still in the top 12 most frequently used languages, and 30% of all data science job listings explicitly list SQL. But industry has done all it can to portray SQL as a barrier to entry, largely through the prevalence of SQL proficiency exams as part of data science job interview processes. The SQL exam is inescapable and scary. And every time I teach this course, I get many understandable requests for more SQL practice questions. Students want to do more SQL to pass the SQL job interview exam, but we've lost something important here: placing the emphasis on the exam misses the fact SQL is one of the very best mechanisms we have in data science for answering questions with data. If you have a database that covers a subject you care about, then you will find that SQL provides you with a working vocabulary to speak to the database as if you are speaking to a sage.

That said, many of you will be facing a SQL exam in the near future. So, you need to practice SQL. A lot. More than you can in this one homework assignment. Fortunately, there are some websites that can help you prepare with deep test banks of relevant SQL-based sample interview questions. Just keep in mind that while the examples you will see on these websites may reasonably strike you as dumb, SQL will be a lot more enjoyable if you find data that matches your interests.

Part a

DataLemur, created by UVA alumnus Nick Singh, offers a range of SQL problems specifically designed for data science interview preparation.

Go to https://datalemur.com/ and sign up for a free account. Then access DataLemur's collection of SQL practice interview questions: https://datalemur.com/questions?

category=SQL. Complete any one question of your choice. Then copy-and-paste the text that appears when your answer is accepted. [4 points]

Accepted

Congrats 🎉 - Share this problem, and your solution, on LinkedIn or Twitter!

In your post, don't forget to tag Nick Singh, so that he can comment on and share your post with his audience of 150k+ followers on LinkedIn and 25k+ followers on Twitter (which will give your post and profile more visibility)!

Part b

LeetCode is well-known for its extensive collection of coding challenges, including a dedicated section for SQL problems. This resource can help you practice a wide variety of SQL queries, from basic to advanced levels.

Go to https://leetcode.com/ and sign up for a free account. Then access Leetcode's collection of SQL practice interview questions: https://leetcode.com/studyplan/top-sql-50/. Correctly complete any one question of your choice. Then describe the first data visualization that appears on the page once your answer is accepted (both the type of visualization and what it means). [4 points]

when successfully finishing a query leetcode will say accepted and print a little graphic analysis of runtime vs other coders.

Part c

StrataScratch offers SQL practice questions sourced from real data science interviews at big companies.

Go to https://www.stratascratch.com and sign up for a free account. Then access StrataScratch's collection of SQL practice interview questions: https://platform.stratascratch.com/coding?code_type=1. Correctly complete any one question of your choice (but make sure there is no lock symbol next to the name, otherwise you have to pay for it, and you should NEVER give your credit card for any homework problems). Then copy-and-paste the text that appears when your answer is accepted. [4 points]

Solved Output Execution time: 0.00546 seconds

View the output in a new tab Your Solution:

Part d

When you take an SQL exam during a job interview, you will not be able to use generative AI to help you. Unless you follow the path of Roy Lee, whose start up aims to build AI mechanisms that give you answers when AI is not available via a browser or IDE. The website's manifesto states plainly "We want to cheat on everything."

Lee was especially motivated to attack this coding-exam-interview ecosystem. A recent article in New York Magazine describes Lee's motivation as follows:

> Then Lee had an idea. As a coder, he had spent some 600 miserable hours on LeetCode, a training platform that prepares coders to answer the algorithmic riddles tech companies ask job and internship candidates during interviews. Lee, like many young developers, found the riddles tedious and mostly irrelevant to the work coders might actually do on the job. What was the point? What if they built a program that hid AI from browsers during remote job interviews so that interviewees could cheat their way through instead?

Please read the New York Magazine article here: https://archive.ph/qIXd0#selection-2129.0-2138.0

Then write a short paragraph describing some of your thoughts and feelings as you read the article. There is no right or wrong answer, so be honest, whether you find yourself firmly on Roy Lee's side, on the other side, or somewhere in the middle, or if you would rather react to the anxieties professors and students express in the article regarding AI in the classroom. Use your own words, please: we care about your opinions, not your grammar.

By the way, whatever your thoughts on the matter may be, Lee's startup just received $5.3 million in venture capital investment. [4 points]

I think that using Chat bots can be beneficial in learning but should not take the place of learning. Students should use it to better understand a topic, trouble shoot, or help them get started with a project, but it should NOT be used as a ways to complete your homework (copy-paste..). At the end of the day this difference is placed on the morals of each individual but it would disadvantage them to have the false impression that they are learning a topic and recieving good grades, however they will be at a disservice and lack of knowledge when they enter the workforce, or have an unrealistic precedence that these ai tools will always be available. I work in the military and there are very frequent cases in which I won't have AI tools readily available so I need to understand the topics in depth.

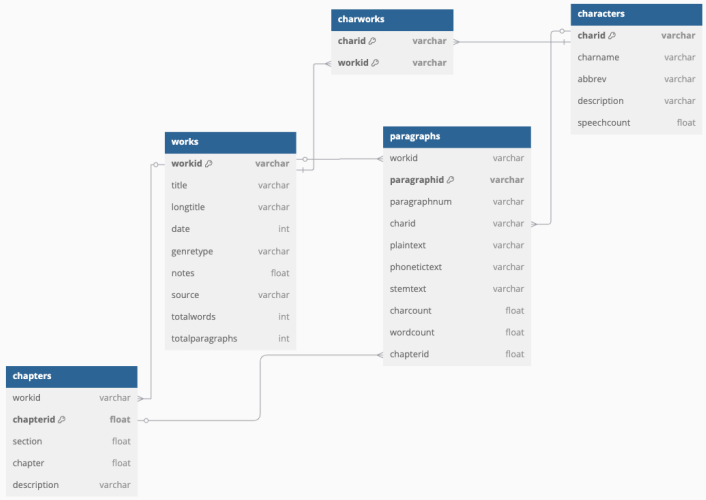
Problem 2

As you saw on DataLemur, Leetcode, and StrataSearch, many of the practice problems regarding SQL refer to vague and general data. That does SQL and database querying in general a major disservice. I like to think about query languages as advanced question-answer engines. We ask a question, and the query combs through all the information contained in the database to get our answer. The problem with the practice problems is that the questions are not especially interesting because the data are artificial.

Now is the winter of our discontent [with SQL] Made glorious summer by this [database assignment about Shakespeare].

In lab 6, we built a database for collected works of Shakespeare.

The ER diagram to help you navigate the data is online, and here:



The five tables are:

works: One row per work authored by Shakespeare, with columns:

- `workid`: (primary key) a unique ID without spaces or special characters for the work
- `title`: the title, such as "Twelfth Night"
- `longtitle`: a longer title, if there is one, such as "Twelfth Night, Or What You Will"
- `date`: year of publication
- `genretype`: `t` is a tragedy, such as *Romeo and Juliet* and *Hamlet*; `c` is a comedy, such as *A Midsummer Night's*

Source: Saturday Morning Breakfast Cereal comics

Dream and *As You Like It*; `h` is a history, such as *Henry V* and *Richard III*; `s` refers to Shakespeare's sonnets; `p` is a narrative (non-sonnet) poem, such as *Venus and Adonis* and *Passionate Pilgrim*

- `notes` : Column for notes from the database maintainer, currently all `NaN`
- `source` : whether the text was originally downloaded from the Moby Project lexicon or Project Gutenberg.
- `totalwords` : Total words in the work
- `totalparagraphs` : Total number of lines of dialogue for plays, or stanzas for poems

characters: One row per character that appears in at least one work by Shakespeare. Some characters, such as Antony or Henry IV, appear in multiple works. Columns:

- `charid` : (primary key) a unique ID for a character
- `charname` : character's name (some characters are different but have the same name, such as the First Musician in Othello and the First Musician in Romeo and Juliet). For poems, the character is "Poet"
- `abbrev` : an abbreviation of the character's name, if needed for reference to some other analyses
- `description` : a longer description of who the character is, if available
- `speechcount` : number of lines of dialogue delivered by the character throughout the works the character appears in

chapters: One row for every unique scene in a play, or for every distinct poem in a collection of poems. Columns:

- `workid` : a unique ID without spaces or special characters for the work
- `chapterid` : (primary key) a unique ID for the scene/poem
- `section` : the scene/poem number
- `chapter` : the act number, if available
- `description` : short description of where the scene takes place, for plays

paragraphs: One row for every line of dialogue that appears in a Shakespeare play, or for every distinct poem in a collection of poems. Columns:

- `workid` : a unique ID without spaces or special characters for the work
- `paragraphid` : (primary key) a unique ID for the line of dialogue/poem
- `paragraphnum` : the position of the paragraph within the ordered list of paragraphs within a chapter
- `charid` : the unique ID of the character delivering the line of dialogue/poem
- `plaintext` : the text of the dialogue/poem
- `phonetictext` : the text of the dialogue/poem in phonetic text, useful for training computers to generate audio of this spoken text
- `stemtext` : the stems of the words in the text, useful for text analyses such as sentiment analysis
- `section` : number of the scene/poem this line appears in
- `chapter` : number of the act this line of dialogue appears in, if a play
- `charcount` : number of characters in the line
- `wordcount` : number of words in the line
- `chapterid` : unique ID for the scene/poem

charworks: One row for every unique combination of character and play. Most characters appear once, but some (such as Antony or Henry IV) appear multiple times. Columns

- `charid` : (primary key) unique ID for the character
- `workid` : (primary key) unique ID for the work

In this problem, we will be posing questions about Shakespeare's work that can be answered using our database, and we will use SQL to get answers to those questions.

In [3]:
```
dotenv.load_dotenv()
POSTGRES_PASSWORD = os.getenv('POSTGRES_PASSWORD')
MONGO_INITDB_ROOT_USERNAME = os.getenv('MONGO_INITDB_ROOT_USERNAME')
MONGO_INITDB_ROOT_PASSWORD = os.getenv('MONGO_INITDB_ROOT_PASSWORD')
```

In [4]:
```
# Test MongoDB
myclient = pymongo.MongoClient(f"mongodb://{MONGO_INITDB_ROOT_USERNAME}:{MONGO_INIT
myclient.list_databases()
```

Out[4]:
```
<pymongo.synchronous.command_cursor.CommandCursor at 0x1fb16cd17f0>
```

In [5]:
```
# Test PostgreSQL
dbserver = psycopg.connect(
    user='postgres',
```

```
    password=POSTGRES_PASSWORD,
    host='localhost',
    port = '5432'
)
dbserver.autocommit = True
```

The Kernel crashed while executing code in the current cell or a previous cell.

Please review the code in the cell(s) to identify a possible cause of the failure.

Click here for more info.

View Jupyter log for further details.

In [ ]:
```
cursor = dbserver.cursor()
```

Part a

Launch the docker containers you used for lab 6 to run the database systems, including PostgreSQL and MongoDB. Load your PostgreSQL password from your .env file, taking care not to expose it in your code. Then use the `create_engine()` method from `sqlalchemy` to create an engine that connects to the PostgreSQL shakespeare database you saved in lab 6. Demonstrate that the connection is successful by using the `pd.read_sql_query()` method to run the following SQL code:

```
SELECT *
FROM works
```

[4 points]

In [ ]:
```
dbms = 'postgresql'
package = 'psycopg'
user = 'postgres'
password = POSTGRES_PASSWORD
host = 'localhost'
port = '5432'
db = 'shakespeare_postgress_db'

engine = create_engine(f"{dbms}+{package}://{user}:{password}@{host}:{port}/{db}")
engine
```

Out[ ]:
```
Engine(postgresql+psycopg://postgres:***@localhost:5432/shakespeare_postgress_db)
```

In [ ]:
```
myquery = '''
SELECT *
FROM works
'''
pd.read_sql_query(myquery, con=engine).head()
```

Out[ ]:

	workid	title	longtitle	date	genretype	notes	source	totalwords	tota
0	12night	Twelfth Night	Twelfth Night, Or What You Will	1599	c	None	Moby	19837	
1	allswell	All's Well That Ends Well	All's Well That Ends Well	1602	c	None	Moby	22997	
2	antonycleo	Antony and Cleopatra	Antony and Cleopatra	1606	t	None	Moby	24905	
3	asyoulikeit	As You Like It	As You Like It	1599	c	None	Gutenberg	21690	
4	comedyerrors	Comedy of Errors	The Comedy of Errors	1589	c	None	Moby	14692	

Now we will use the Shakespeare DB to answer some questions that come from the academic research on Shakespeare's work. The best way to write a SQL query is one step at a time, looking at the output as you go along. The first few problems will guide you through these steps, but later problems will leave the step-by-step construction of a query up to you.

Part b

Lucas Erne (2013) argues that Shakespeare wrote his plays for the stage and also for the written page, abridging the works for stage adaptation. This thesis is controversial because Shakespeare has been viewed as a playwright writing for live audiences rather than as an author writing for posterity, and Erne challenges that conception. One theory is that as Shakespeare experienced greater levels of success, he viewed himself more through the lens of his own legacy, and wrote later works in a more literary way than earlier works, producing works of greater length.

We'll write a SQL query that examines whether early, middle, or late Shakespeare works are longer, to test this theory.

Step i

Write and execute a SQL query that pulls the `title` and `date` columns from the `works` table, and renames `date` to `year`. [4 points]

In [ ]:
```
myquery = '''
SELECT title, date AS year
FROM works
```

```
'''
pd.read_sql_query(myquery, con=engine).head()
```

Out[ ]:

	title	year
0	Twelfth Night	1599
1	All's Well That Ends Well	1602
2	Antony and Cleopatra	1606
3	As You Like It	1599
4	Comedy of Errors	1589

Step ii

Build on the query you wrote in step (i) by using the `CASE` clause to create a new column named `era` that equals "early" when `date` is earlier than 1600, "middle" if `date` is between 1600 and 1607, and "late" if `date` is after 1607. (Note: SQL doesn't allow you to use the new name `year` instead of the old name `date` in the rest of the SQL code, unless you are using a subquery or a common table expression.) [4 points]

In [ ]:
```
myquery = '''
SELECT title, date AS year,
CASE
    WHEN date < 1600 THEN 'early'
    WHEN date BETWEEN 1600 AND 1607 THEN 'middle'
    WHEN date > 1607 THEN 'late'
    END AS era
FROM works
'''
pd.read_sql_query(myquery, con=engine).head()
```

Out[ ]:

	title	year	era
0	Twelfth Night	1599	early
1	All's Well That Ends Well	1602	middle
2	Antony and Cleopatra	1606	middle
3	As You Like It	1599	early
4	Comedy of Errors	1589	early

Step iii

Build on to the SQL query you wrote for step (ii) by using the `GROUP BY` clause to collapse the data to one row per value of `era`. Use the `AVG()` function to take the average of `totalwords` within each value of `era`, and rename the averaged column `average_total_words`. Also, because we don't have a way to summarize across all of the

titles or publication years within each era, remove `title` and `date` from the `SELECT` clause.

Based on the output, is the hypothesis that later works have longer lengths supported, partly supported, or not supported? (Generally speaking that is, no need for a formal hypothesis test here.) [4 points]

In [ ]:
```
myquery = '''
SELECT
CASE
    WHEN date < 1600 THEN 'early'
    WHEN date BETWEEN 1600 AND 1607 THEN 'middle'
    WHEN date > 1607 THEN 'late'
    END AS era,
    AVG(totalwords) AS average_total_words
FROM works
GROUP BY era
'''
#okay but like why does end as need a comma after it sometimes and sometimes not???
pd.read_sql_query(myquery, con=engine).head()
```

Out[ ]:

	era	average_total_words
0	late	18905.571429
1	early	20328.500000
2	middle	21999.416667

The longer works are longer than his earlier works. This could go along with hypothesis.

Part c

Colyvas, Egan, and Craig (2023) argue that Shakespeare was influenced by contemporary literary trends and like his peers began using shorter speeches after 1600. We can test these theories by using SQL to measure whether later works tend to have longer or shorter lines of dialogue than earlier works. We'll construct this query step-by-step.

Step i

Join the `works` table to the `paragraphs` table (an inner join is fine because there should be no works without paragraphs, and no paragraphs without a work), and keep all columns from both tables. [4 points]

In [ ]:
```
myquery = '''
SELECT *
FROM WORKS
INNER JOIN paragraphs
    ON paragraphs.workid = works.workid
'''
```

```
pd.read_sql_query(myquery, con=engine).head()
```

Out[ ]:

	workid	title	longtitle	date	genretype	notes	source	totalwords	totalparagraphs
0	12night	Twelfth Night	Twelfth Night, Or What You Will	1599	c	None	Moby	19837	1031
1	12night	Twelfth Night	Twelfth Night, Or What You Will	1599	c	None	Moby	19837	1031
2	12night	Twelfth Night	Twelfth Night, Or What You Will	1599	c	None	Moby	19837	1031
3	12night	Twelfth Night	Twelfth Night, Or What You Will	1599	c	None	Moby	19837	1031
4	12night	Twelfth Night	Twelfth Night, Or What You Will	1599	c	None	Moby	19837	1031

Step ii

Build on the query you wrote in step (i) by keeping only the `title` column from the `works` table, the `plaintext` and `wordcount` columns from the `paragraphs` table, and creating a new column named `era` that is equal to "before 1600" for values of `date` before 1600, and "1600 or later" for values of `date` greater than or equal to 1600. But don't save the `date` column in the output. [4 points]

In [ ]:

```
myquery = '''
SELECT w.title, p.plaintext, p.wordcount,
CASE
    WHEN w.date < 1600 THEN 'before 1600'
    WHEN w.date >= 1600 THEN '1600 or later'
    END AS era
FROM WORKS w
INNER JOIN paragraphs p
    ON p.workid = w.workid
'''
```

```
pd.read_sql_query(myquery, con=engine).head()
```

Out[ ]:

	title	plaintext	wordcount	era
0	Twelfth Night	[Exit]	1.0	before 1600
1	Twelfth Night	[Sings]\n[p]When that I was and a little tiny ...	124.0	before 1600
2	Twelfth Night	[Exeunt all, except Clown]\n	4.0	before 1600
3	Twelfth Night	Pursue him and entreat him to a peace:\n[p]He ...	70.0	before 1600
4	Twelfth Night	He hath been most notoriously abused.\n	6.0	before 1600

Step iii

Build off of the query you wrote for step (ii) by collapsing the data to one row per value of `era` with a column containing the average `wordcount` across paragraphs in that era. Remove any columns that cannot be meaningfully collapsed to one row per era.

Based on your result, does the evidence support the hypothesis that Shakespeare wrote shorter lines of dialogue starting in 1600? (No need for a formal hypothesis test) [4 points]

In [ ]:
```
myquery = '''
SELECT
CASE
    WHEN w.date < 1600 THEN 'before 1600'
    WHEN w.date >= 1600 THEN '1600 or later'
    END AS era,
    AVG(wordcount) as average_wordcount
FROM WORKS w
INNER JOIN paragraphs p
    ON p.workid = w.workid
GROUP BY era
'''

pd.read_sql_query(myquery, con=engine).head()
```

Out[ ]:

	era	average_wordcount
0	before 1600	25.917012
1	1600 or later	23.815456

Part d

Shakespeare's plays were grouped into the broader genres of histories, tragedies, and comedies by a group of his colleagues for the publication of 36 of his plays in the First Folio in 1623. Whissell (2007) analyzes how Shakespeare used word choice, specifically when conveying emotion and imagery, to connote genre. Specifically, Whissell finds that comedies

more frequently than tragedies use words associated with "pleasantness", such as bright, kinder, rapt, satisfied, success, virtues, wise, hero, love, hope, and relief. (The full dictionary of works associated with pleasantness, activation, and imagery is here)

Write a SQL query that calculates whether each line of dialogue from every Shakespeare work contains one of these words associated with pleasantness, then calculates the proportion of total lines of dialogue from each work that contain a pleasantness word, then averages these proportions across genres to test whether there is a difference between tragedies and comedies. We will again do this together, step-by-step (though the steps are bigger ones in this problem).

Step i

Write a query to keep the `workid` and `plaintext` columns from the `paragraphs` table. Also, create a new column called `pleasant_line` that is equal to 1 if any of the words **bright**, **kinder**, **rapt**, **satisfied**, **success**, **virtues**, **wise**, **hero**, **love**, **hope**, or **relief** appear in the line. A few notes:

- To search for specific text such as "bright" within the `plaintext` column, use the `LIKE '%%bright%%'` syntax, as shown in section 7.6.6. of the textbook.
- The SQL text search function is case-sensitive, but we want to identify these words whether or not they are capitalized. To remove the upper-case letters from `plaintext` while searching for text, use `LOWER(plaintext) LIKE '%%bright%%'`.
- To search for multiple terms, you will have to string the search syntax for each term together with a series of `OR` clauses: `LOWER(plaintext) LIKE '%%bright%%' OR LOWER(plaintext) LIKE '%%kinder%%' OR . . .`.
- The default output of a `LIKE` clause is a logical type True/False column. To turn this column into 0s for False and 1s for True (to enable arithmetic operations later) wrap the column inside `CAST(... AS INT)` to convert it to an integer-type column.

[4 points]

In [ ]:
```
myquery = '''
SELECT workid, plaintext,
CAST( LOWER(plaintext) LIKE '%%pleasant%%' OR
    LOWER(plaintext) LIKE '%%bright%%' OR
    LOWER(plaintext) LIKE '%%kinder%%'  OR
    LOWER(plaintext) LIKE '%%rapt%%'  OR
    LOWER(plaintext) LIKE '%%satisfied%%'   OR
    LOWER(plaintext) LIKE '%%success%%'  OR
    LOWER(plaintext) LIKE '%%virtues%%'  OR
    LOWER(plaintext) LIKE '%%wise%%'  OR
    LOWER(plaintext) LIKE '%%hero%%'  OR
    LOWER(plaintext) LIKE '%%love%%'  OR
    LOWER(plaintext) LIKE '%%hope%%'  OR
    LOWER(plaintext) LIKE '%%relief%%'
AS INT) AS pleasant_line
FROM paragraphs
```

```
'''

pd.read_sql_query(myquery, con=engine).head()
```

Out[ ]:

	workid	plaintext	pleasant_line
0	12night	[Enter DUKE ORSINO, CURIO, and other Lords; Mu...	0
1	12night	If music be the food of love, play on;\n[p]Giv...	1
2	12night	Will you go hunt, my lord?\n	0
3	12night	What, Curio?\n	0
4	12night	The hart.\n	0

Step ii

Building off of the query you wrote for part (i), collapse the data to one row per work with the average value of `pleasant_line` (the average of a column of 0s and 1s is the proportion of rows that have a 1). [4 points]

In [ ]:

```
myquery = '''
SELECT  workid,
AVG(
CAST( LOWER(plaintext) LIKE '%%pleasant%%' OR
    LOWER(plaintext) LIKE '%%bright%%' OR
    LOWER(plaintext) LIKE '%%kinder%%'  OR
    LOWER(plaintext) LIKE '%%rapt%%'  OR
    LOWER(plaintext) LIKE '%%satisfied%%'   OR
    LOWER(plaintext) LIKE '%%success%%'  OR
    LOWER(plaintext) LIKE '%%virtues%%'  OR
    LOWER(plaintext) LIKE '%%wise%%'  OR
    LOWER(plaintext) LIKE '%%hero%%'  OR
    LOWER(plaintext) LIKE '%%love%%'  OR
    LOWER(plaintext) LIKE '%%hope%%'  OR
    LOWER(plaintext) LIKE '%%relief%%'
AS INT)) AS pleasant_proportion
FROM paragraphs
GROUP BY workid

'''

pd.read_sql_query(myquery, con=engine).head()
```

Out[ ]:

	workid	pleasant_proportion
0	12night	0.083414
1	allswell	0.090732
2	antonycleo	0.053571
3	asyoulikeit	0.141055
4	comedyerrors	0.045386

Step iii

Using a subquery as shown in section 7.6.8 of the textbook, or a common table expression as shown here, treat the output of your query from step (ii) as a new, temporary table. Join this table with the `works` table, and collapse the data to one row per genre with the average across works of the proportions you calculated in step (ii).

Based on your result, does the evidence support the hypothesis that comedies have higher use of pleasant-words than tragedies? (Again, no need for any formal hypothesis test or analysis beyond the SQL query output).

[Note: in the steps laid out in this problem, you might reasonably think we can first join the parapgraph and works tables, then create the `pleasant_line` average, so as to avoid using a subquery or a common table expression. The reason why this approach doesn't work is because we've renamed a column to `pleasant_line` from a very complicated default name, but this alias name is not understood by PostgreSQL until the query is executed, so we would have to use the complicated name when collapsing the data to genres. Instead, by using a subquery or CTE, the first part of the query is executed and saved, along with the column alias, for use in the second part of the query.]

[8 points]

In [ ]:

```
myquery = '''
WITH pleasant_by_work AS (
    SELECT
        workid,
        AVG(
            CAST(
                LOWER(plaintext) LIKE '%%pleasant%%' OR
                LOWER(plaintext) LIKE '%%bright%%' OR
                LOWER(plaintext) LIKE '%%kinder%%' OR
                LOWER(plaintext) LIKE '%%rapt%%' OR
                LOWER(plaintext) LIKE '%%satisfied%%' OR
                LOWER(plaintext) LIKE '%%success%%' OR
                LOWER(plaintext) LIKE '%%virtues%%' OR
                LOWER(plaintext) LIKE '%%wise%%' OR
                LOWER(plaintext) LIKE '%%hero%%' OR
                LOWER(plaintext) LIKE '%%love%%' OR
                LOWER(plaintext) LIKE '%%hope%%' OR
```

```
                LOWER(plaintext) LIKE '%%relief%%'
                AS INT
            )
        ) AS pleasant_proportion
    FROM paragraphs
    GROUP BY workid
)

SELECT
    w.genretype,
    AVG(p.pleasant_proportion) AS avg_pleasant_proportion
FROM pleasant_by_work p
JOIN works w
    ON p.workid = w.workid
GROUP BY w.genretype
ORDER BY avg_pleasant_proportion DESC
'''

pd.read_sql_query(myquery, con=engine).head()
```

Out[ ]:

	genretype	avg_pleasant_proportion
0	s	0.707792
1	p	0.303818
2	c	0.101053
3	h	0.075128
4	t	0.075009

Part e

Champion (1976) analyzes the soliloquies in Hamlet. A soliloquoy is a speech that a character gives in which they speak to themselves. These speeches represent a character's inner thoughts and internal monologue. In Hamlet, Champion notes that the soliloquoys represent "Hamlet's inner struggle. The devices of internalization, for example, are far more extensive than in the earlier tragedies. More important, the clue to Hamlet's personality is to be found in the quite unique manner in which the playwright utilizes these devices to reveal the complexity of this inner man."

Write a SQL query that displays the longest speech delivered by Hamlet. Then, in writing, identify part of this speech that demonstrates Hamlet's inner struggle.

This time you are on your own to take the steps you need. But please remember that the best way to write a SQL query is to take it step-by-step.

A few notes:

- In this case, `wordcount` in the paragraphs table measures the length of speeches, so use this column. But in general if we did not have this column, we could use the

`LENGTH()` clause in SQL to count the number of characters in each text value.

- To see the maximum value of a column, you can sort with `ORDER BY` with the `DESC` clause, then use `LIMIT 1` to show only the first value.
- The output of `pd.read_sql_query()` is stored as a Pandas dataframe. To extract the first value of a column named `plaintext` and to print it in a readable way, you can write: `print(pd.read_sql_query(myquery, con=engine)['plaintext'][0])` in Python, outside the SQL code.

[8 points]

In [ ]:
```
myquery = '''
SELECT *
FROM paragraphs
WHERE MAX(LENGTH(wordcount))
'''
```

Part f

Berry (2016) argues that Shakespeare's plays are "powerfully shaped by their sense of place." That is, the choice of setting, whether a castle or palace, or a house or a garden, sets a tone which impacts the narrative and its effect on the audience.

Write a SQL query that generates a table with one row for each of Shakespeare's works, and two columns: the title of each work, and the count of the number of scenes in each work that take place in a castle or a palace, sorted by count in descending order.

Remember, you don't have to know the whole solution from the outset. Take it step-by-step and get closer and closer to the right answer until you have it. [12 points]

In [ ]:
```

```

Problem 3

The following file contains JSON formatted data of the official English-language translations of every constitution that was in effect in the world as of 2013:

In [ ]:
```
const = requests.get("https://github.com/jkropko/DS-6001/raw/master/localdata/const
const_json = json.loads(const.text)
pd.DataFrame.from_records(const_json)
```

Out[ ]:

	text	country	adopted	revised	reinstated	democracy
0	'Afghanistan 2004 Preamble \nIn the na...	Afghanistan	2004	NaN	NaN	0.372201
1	'Albania 1998 (rev. 2012) Preamble \nWe...	Albania	1998	2012.0	NaN	0.535111
2	'Andorra 1993 Preamble \nThe Andorran P...	Andorra	1993	NaN	NaN	NaN
3	'Angola 2010 Preamble \nWe, the people ...	Angola	2010	NaN	NaN	0.315043
4	'Antigua and Barbuda 1981 Preamble \nWH...	Antigua and Barbuda	1981	NaN	NaN	NaN
...	...	...	...	...	...	...
140	'Uzbekistan 1992 (rev. 2011) Preamble \...	Uzbekistan	1992	2011.0	NaN	0.195932
141	'Viet Nam 1992 (rev. 2013) Preamble \nI...	Viet Nam	1992	2013.0	NaN	0.251461
142	'Yemen 1991 (rev. 2001) PART ONE. THE FOUN...	Yemen	1991	2001.0	NaN	0.125708
143	'Zambia 1991 (rev. 2009) Preamble \nWE,...	Zambia	1991	2009.0	NaN	0.405497
144	'Zimbabwe 2013 Preamble \nWe the people...	Zimbabwe	2013	NaN	NaN	0.315359

145 rows × 6 columns

The text of the constitutions are available from the Wolfram Data Repository. I also included scores that represent the level of democratic quality in each country as of 2016. These scores are compiled by the Varieties of Democracy (V-Dem) project. Higher scores indicate greater levels of democratic openness and competition.

Part a

Connect to your local MongoDB server and create a new collection for the constitution data. Use `.delete_many({})` to remove any existing data from this collection, and insert the data in `const_json` into this collection. [4 points]

In [ ]:
```
#connect!!!
mongo_user = os.getenv('MONGO_INITDB_ROOT_USERNAME')
mongo_password = os.getenv('MONGO_INITDB_ROOT_PASSWORD')
host = 'localhost'
port = 27017
myclient = pymongo.MongoClient(f"mongodb://{mongo_user}:{mongo_password}@{host}:{po
```

```
constitution = myclient["history"]
collection = constitution["constitution"]

collection.delete_many({})

collection.insert_many(const_json)

print(collection.find_one().keys())
```

Part b

Use MongoDB queries to produce dataframes with the following restrictions:

- The country, adoption year, and democracy features (and not `_id`, text, revised, or reinstated) for countries with constitutions that were written after 1990
- The country, adoption year, and democracy features (and not `_id`, text, revised, or reinstated) for countries with constitutions that were written after 1990 AND have a democracy score of less than 0.5
- The country, adoption year, and democracy features (and not `_id`, text, revised, or reinstated) for countries with constitutions that were written after 1990 OR have a democracy score of less than 0.5

[4 points]

In [ ]:
```
# Strongly dislike the change in syntax here. its a lot harder to read
```

In [ ]:
```
def mongo_read_query(col, q):
    qtext = dumps(col.find(q))
    qrec = loads(qtext)
    qdf = pd.DataFrame.from_records(qrec)
    return qdf
```

In [ ]:
```
wha_clean
```

		Industry	Size	organization_type	payment_premium	premium_change
0	Hospital_Worksites	Large	Nonprofit	Partial	Larger
1	Hospital_Worksites	Large	Nonprofit	Partial	Same
2	Hospital_Worksites	Large	Nonprofit	Full	Same
3	Hospital_Worksites	Medium	Private	Full	Smaller
4	Hospital_Worksites	Medium	Nonprofit	Full	Same
...	...	...	...	...	...
2838	Public_Administration	Medium	State govt	Full	Same
2839	Public_Administration	Medium	State govt	Partial	Same
2840	Public_Administration	Large	State govt	Partial	Same
2841	Public_Administration	Large	State govt	Partial	Same
2842	Public_Administration	Large	State govt	Partial	Same

2843 rows × 17 columns

In [ ]:
```
#The country, adoption year, and democracy features (and not `_id`, text, revised,
myquery = {
    "adopted": {"$gt": 1990}
}

mongo_read_query(collection, myquery)
```

In [ ]:
```
# The country, adoption year, and democracy features (and not `_id`, text, revised,
myquery = {
    "adopted": {"$gt": 1990},
    "democracy": {
        "$lt": 0.5
    }
}

mongo_read_query(collection, myquery)
```

In [ ]:
```
#Use orrr
#The country, adoption year, and democracy features (and not `_id`, text, revised,

myquery = {
    "$or": [
        {"adopted": {"$gt": 1990}},
        {"democracy": {
            "$lt": 0.5
        }}
    ]
}
```

```
mongo_read_query(collection, myquery)
```

Part c

According to the Varieties of Democracy project, Hungary has become less democratic over the last few years, and can no longer be considered a democracy. Update the record for Hungary to set the democracy score at 0.4. Then query the database to extract the record for Hungary and display the data in a dataframe. [4 points]

In [ ]:
```
myquery = {
    "country": "Hungary"
}
update_operation = {
    "$set": {
        "democracy": 0.4
    }
}

collection.update_one(myquery, update_operation)
```

In [ ]:
```
myquery = {
    "country": "Hungary"
}

mongo_read_query(collection, myquery)
```

Part d

Set the `text` field in the database as a text index. Then query the database to find all constitutions that contain the exact phrase "freedom of speech". Display the country name, adoption year, and democracy scores in a dataframe for the constitutions that match this query. [4 points]

In [ ]:
```
collection.create_index([("text", "text")]) #wut me no understando

myquery = collection.find(
    {"$text": {"$search": "\"freedom of speech\""}},
    {"_id":0, "text": 0, "revised":0, "reinstated":0}
)

constitutionz = pd.DataFrame(list(myquery))
constitutionz
```

Part e

Use a query to search for the terms "freedom", "liberty", "legal", "justice", and "rights". Generate a text score for all of the countries, and display the data for the countries with the top 10 relevancy scores in a dataframe. [4 points]

In [ ]:
```
terms = "freedom liberty legal justice rights"

myquery = collection.find(
    {"$text": {"$search": terms}},
    {"_id":0, "text": 0, "revised":0, "reinstated":0, "score": {"$meta": "textScore
)

results = myquery.sort([("score", {"$meta": "textScore"})]).limit(10)
countries = pd.DataFrame(list(results))
countries
```

Question 4

Once you are finished working with databases, clear up the space on your computer by going to the terminal that you used to launch the Docker containers, press CONTROL + C on your keyboard to stop the containers, then type `docker compose down` to disconnect the volumes and networks. It's a good idea to make a practice out of doing these steps when you finish working with databases.

This problem isn't graded, and no need to write anything. But please do this anyway.

can't do it if my computer broken :/