ScienceForums.Net data analysis

(2 points.) To start, run `python new-homework.py [language] sfn`

Background. To test your SQL skills, I have extracted a sample of discussion topics and posts at ScienceForums.Net, a discussion forum that has been open since 2002. SFN covers a range of discussion topics, including standard science topics like physics and chemistry but also a general discussion section, politics, and a rather controversial religion section.

SFN has had about 800,000 posts since 2002. Most members are not professional scientists, though there are some "resident experts" who are academics or industry scientists. There is a team of volunteer moderators who keep discussions from getting out of hand, deleting posts or closing discussions as necessary. (Generally they try to post warnings rather than deleting anything.)

One section deserves special mention. For some reason, online science forums attract the attention of crackpots who believe they can prove the theory of relativity wrong or write a new Theory of Everything. SFN gets quite a few of them, and their posts get moved to the Speculations section. They tend to refuse to listen to evidence or argument, so eventually their topics get closed, but in the mean time they attract the attention of many other members who enjoy an easy smackdown. So you may notice some unusual trends in Speculations.

The dataset comes as three files, `topics.csv`, `posts.csv.gz`, and `forums.csv.gz`, provided in the `sfn/Data/` directory. (The `.gz` files are compressed; the `gunzip` command can decompress them.) In a discussion forum, each post is contained within a topic of discussion, and each discussion is contained in a forum—basically a category or section of the website. The files contain some extraneous columns you don't need to worry about, but here are the key fields. In `posts.csv`,

pid A numeric sequential post ID.

author_id The ID of the user who created the post.

post_date The time the post was posted, as a Unix timestamp (number of seconds since January 1 1970). (The Postgres `to_timestamp` function can automatically convert this to a Postgres timestamp.)

topic_id The ID of the topic this post was created in.

queued If this value is 2, the post was hidden from public view.

In `topics.csv`,

tid The topic ID.

title The title of the topic, as free-form text. (You will notice some spam, like "4949 nexium stomach ache.")

state If "closed", the topic has been locked.

posts The number of posts made in the discussion. May be out of sync with `posts.csv` for various reasons, so you should not rely on this field.

start_date Date the discussion was started, again as a Unix timestamp.

last_post Date of the most recent post in the discussion.

views Number of times the discussion has been viewed.

forum_id The ID of the forum the discussion was posted in.

Note that **some topics have been removed**, so there will be posts whose topic IDs do not match any topics. (These are posts in private sections, such as where moderators coordinate their work, where revealing the topic title may be problematic.) You may omit these posts from the database.

Finally, in `forums.csv`,

id The forum ID.

name The public name of the forum.

parent_id Some forums are contained inside other forums. This is the ID of the parent.

Task. Write a schema for a set of database tables containing the post, topic, and forum data. You do not need to include every field from the CSV files, just the important ones highlighted above. You may use as many or as few tables as you'd like, as long as you remember the DRY principle and do not make them too cumbersome to use. Write your `CREATE TABLE` commands in a file, `schema.sql`, and submit this file with your assignment.

Next, create a database with the tables you specified. Write a script to load the CSV files into your database, doing any necessary conversions or cleaning. R users may want to use RPostgreSQL; Python users may want Psycopg. Interfaces are available for most other languages as well.

Your script needs to connect to our server `sculptor.stat.cmu.edu` to get database access. There are instructions on the course website: `https://36-750.github.io/databases/sql-code/`

Once you have loaded your data, write a set of queries that do the following:

1. Return the title, start date, last post date, and forum name for every posted topic.
2. Return a ranked list of forums, sorted by the average length (last post - start date) of the topics contained in them. The list should contain the forum ID, name, and average length in days.
3. Return a ranked list of member IDs and post counts, sorted by their total number of posts.
4. Rank the forums by the proportion of topics they contain with 10 or more posts. Return the forum ID, name, and proportion.
5. Compare the average number of posts in locked topics to the number in open topics.

These queries should be pure SQL, using the functions and operators provided by Postgres. You should not need to write R, Python, or any other code. PostgreSQL has extensive documentation on SQL syntax, data types, schema definition, built-in functions, and everything else you will need to complete this assignment.

Examples. To load the CSV files into a database, you'll need to use your programming language's Postgres library. These libraries offer many useful features. One of the most important is ***prepared statements***: instead of building a query by concatenating together strings of text, we can parametrize the query and provide values for the database system.

This may seem less convenient, but consider an example in Python using Psycopg:

```
conn = psycopg2.connect(database='yourusername', user='yourusername',
                        password='yourpassword', host='sculptor.stat.cmu.edu')
cur = conn.cursor()

cur.execute("INSERT INTO foo (bar, baz, spam) "
            "VALUES (%(bar)s, %(baz)s, %(spam)s)",
            {'bar': calculate_thingy(), 'baz': 42, 'spam': user_input})
```

The `%(bar)s` syntax tells Psycopg to create a placeholder for a value named `bar`. We then provide the value in a dictionary passed as the second argument to `execute`. The RPostgreSQL package provides the *`sqlInterpolate`* function to do the same thing; examples are given in the lecture notes.

This is much better than the alternate approach, using string concatenation:

```
cur.execute("INSERT INTO foo (bar, baz, spam) "
            "VALUES (" + calculate_thingy() + ", 42, " + user_input + ")")
```

The danger with string concatenation is that variables like `user_input` may contain SQL syntax. For example, if `user_input` is the string `"7); DROP TABLE foo; --"`, then inserting it into the SQL string will cause your data to be deleted.[1] The parametrized version does not have this problem, because Postgres parses the query *before* inserting the user-supplied values into it, and understands that `%(spam)s` is a single value and not part of the SQL query.

Additionally, parametrizing a query you run many times is more efficient. When you first provide the query to Postgres, it develops a "query plan" specifying which indices it will use, what strategies it will use to handle joins, and so on, using a complicated set of heuristics to find the most efficient strategy. If we prepare the statement once and provide many sets of parameter values, Postgres only has to plan the query once. This feature is provided by the *`executemany`* method in Psycopg, or by similar functions in most other database wrapper libraries.

Requirements.

- ☐ Write a careful schema for the provided data, using foreign keys, constraints, and Postgres data types as appropriate.
- ☐ Supply code to load the CSV files into your schema using prepared statements.
- ☐ Supply the SQL queries that answer the provided questions. Paste them into a `.sql` file or text file and include comments describing how they work when necessary.

[1] This problem is known as *SQL injection*, and is a major source of security vulnerabilities in websites. There is, of course, a relevant xkcd.