University of Toronto
CSC343, Fall 2021

Assignment 2

Original due date: Thursday, November 4th, before 8pm
Extended to: Sunday, November 7th, before 8pm

Part 3: Embedded SQL with psycopg2

For Part 3, you will complete methods in **a2.py**. You can find the documentation for psycopg2 here: `https://www.psycopg.org/docs/`

The code provides infrastructure for a recommender system. It includes two different methods that can recommend items for purchase: `recommendGeneric` makes recommendations that are based on item ratings and are not tailored to a particular customer's tastes, while `recommend` tailors its recommendations to a customer by comparing that customer's ratings to those of other people with similar tastes. Here is how method `recommend` is to work:

1. Find the curator whose taste is most like the customer by finding the one whose ratings of the 2 most-sold items in each category are most similar to the customer's own ratings on these same items.
2. Recommend the items (any items, not just those that are among the most sold) that this curator rated highly and that the customer hasn't bought.

(See the method comments in a2.py for further details on both of the recommend methods.)

If you consider that recommendations will be given very often and that computing definitive ratings will be very costly, this approach doesn't seem practical. However, do the definitive ratings really have to be recalculated every single time a recommendation is given in order to make good recommendations? Probably not. It may well be that working from what were the definitive ratings as of midnight last night yields good enough recommendations. Or if we want to be improve recommendations, perhaps we should base them on what were the definitive ratings at most an hour ago. So we can save a "snapshot" of the definitive ratings periodically and make our recommendations based on those. Having to compute the definitive ratings only periodically rather than once per recommendation would certainly save a lot of computation!

Method `recommend` uses a snapshot of the definitive ratings that is stored in a table called `DefinitiveRatings`. This table gets [re-]built every time `repopulate` is called, and a recommender system that is built based on our Recommender class has the choice of whether to do so once a day, once an hour, every so-many recommendations, or based on some other strategy. Similarly, we keep a snapshot of the most-sold items in a table called `PopularItems`, which is updated each time `repopulate` is called.

You may have noticed that method `recommend` needs to know who the curators are. They can be identified using the view you defined in Question 3. Make sure that you have imported q3.sql when you test the recommend method.

Method `recommend` will need to use class `RatingsTable`, which we defined for you in `ratings.py`. See the docstring examples for how to use this simple class. You will find the function `find_similar_curator`, which we have provided, is also helpful for method `recommend`.

Your task

Complete the methods that we have documented in the starter code in `a2.py`.

1. `recommend_generic`: Recommends items to the customer based on item ratings.
2. `recommend`: Recommends items to the customer based on the ratings of the most similar curator.
3. `repopulate`: Updates the "snapshot" of data in tables `DefinitiveRatings` and `PopularItems` by emptying them out, recomputing the definitive ratings and popular items based on the current data in the database, and inserting these into `DefinitiveRatings` and `PopularItems`. A helper method for `recommend`.

I don't want you to spend a lot of time learning Python for this assignment, so feel free to ask Python-specific questions as they come up.

Important Guidelines

- You may not use standard input in the methods that you are completing. Doing so will result in the auto-tester timing out, causing you to receive a **zero** on that method. (You can use standard input in any testing code that you write outside of these methods, however.)
- Do not change any of the code provided. In particular, you must not change the header of any of the methods we've asked you to implement.
- Do not call `connect_db()` or `disconnect_db()` in the other methods we ask you to implement; you can assume that they will be called before and after, respectively, any other method calls.
- All of your code must be written in `a2.py`. This is the only file you may submit for this part.
- You are welcome to write helper methods to maintain good code quality.
- Within any of your methods, you are welcome to define views to break your task into steps. Drop those views before the method returns, or otherwise a subsequent call to the method will raise an error when it tries to define a view that already exists.
- Your methods should do only what the docstring comments say to do. In some cases there are other things that might have made sense to do but that we did not specify, in order to simplify your work.
- If behaviour is not specified in a particular case, we will not test that case.

How your work for Parts 1, 2 and 3 will be marked

This assignment will be entirely marked via auto-testing. Your mark for each part will be determined by the number of test cases that you pass. To help you perform a basic test of your code (and to make sure that your code connects properly with our testing infrastructure), approximately a week before the due date we will provide some self-tests that you can run through MarkUs. Watch for an announcement on Quercus.

We will of course test your code on a more thorough set of test cases when we grade it, and you should do the same.

Some advice on testing your code

Testing is a significant task, and is part of your work for A2. You'll need a dataset for each condition / scenario you want to test. These can be small, and they can be minor variations on each other. I suggest you start your testing for a given query by making a list of scenarios and giving each of them a memorable name. Then create a dataset for each, and use systematic naming for each file, such as `q1-no-items-reviewed`. Then to test a single query, you can:

1. Import the schema into psql (to empty out the database and start fresh).
2. Import the dataset (to create the condition you are testing).
3. Then import the query and review the results to see if they are as you expect.

Repeat for the other datasets representing other conditions of interest for that query.

Testing for part 3 can be done in a similar way to the "Coordinating" part of the Embedded exercises posted on our Lectures page. I recommend being very organized, as described above. In this case, to test a method on a particular dataset:

1. Have two windows logged in to dbsrv1.
2. In window 1, start psql and import the schema (to empty out the database and start fresh) and then the dataset you are going to test with.
3. In window 2 (remember, this is on dbsrv1), modify the main block of your a2 program so that it has an appropriate call to the method you are about to test. Then run the Python code.
4. Back in psql in window 1, check that the state of your tables is as you expect.

Don't forget to check the method's return value too.

You may find it helpful to define one or more functions for testing. Each would include the necessary setup and call(s) to your method(s). Then in the main block, you can include a call to each of your testing functions, comment them all out, and uncomment-out the one you want to run at any given time.

Your main block and testing functions, as well as any helper methods you choose to write, will have to effect on our auto-testing.