University of Toronto
CSC343, Winter 2026

Assignment 2

Warmup due: Wed, February 25, before 3pm
Full assignment due: Wed, March 18, before 3pm

This is the handout for the full Assignment 2. The Warmup assignment is just a subset of it: Query 1 (Unrated Products) from Part 1 and Update 1 (SALE!SALE!SALE!) from Part 2.

Learning Goals

The purpose of this assignment is to give you practise writing complex stand-alone SQL queries, experience using psycopg2 to embed SQL queries in a Python program, and a sense of why a blend of SQL and a general-purpose language can be the best solution for some problems.

By the end of this assignment you will be able to:

- read and interpret a novel schema written in SQL
- write complex queries in SQL
- design datasets to test a SQL query thoroughly
- quickly find and understand needed information in the postgreSQL and psycopg2 documentation
- embed SQL in a high-level language using psycopg2
- recognize limits to the expressive power of standard SQL

Please read this assignment thoroughly before you proceed. Failure to follow instructions can affect your grade.

We will be testing your code in the **CS Teaching Labs environment** using PostgreSQL. It is your responsibility to make sure your code runs in this environment before the deadline! **Code which works on your machine but not on the CS Teaching Labs will not receive credit**.

Introduction

For this assignment, you will build components for a recommender system for online shopping. You will work with a database that can store information about customers, online orders they make, and what items are included in each order. (Note: Throughout this assignment, we use the terms "order" and "purchase" interchangeably.) The database also stores customer reviews, which have two parts: a numeric rating and a written comment. Customers do not have to give a review, but if they do, the numeric rating is required; the written comment is optional. Customers can also rate each other's reviews as either helpful or unhelpful.

In Part 1, you will write interactive queries on the database. and in Part 2 you will write SQL to perform insert, update, and delete operations on the database.

For part 3 of the assignment, you will create some infrastructure that could be used to recommend items to customers based on the reviews and the helpfulness ratings in the database. Interactive queries are inadequate for a good recommender system. Recommender systems are a huge area of interest in both research and industry. The ACM Conference on Recommender Systems (`http://recsys.acm.org`), is a good place to get a sense of some of the techniques that are used. You'll see phrases like "multi-value probabilistic matrix factorization" and "random walk based entity ranking." Here's a very simple technique for generating recommendations for person p: build a matrix with the recommendations of various other users for various items, find among these users the person q with the most similar taste to p (comparing a vector of p's own recommendations to the recommendations of users represented in the matrix), and then report back the favourite items of q. It's hard to imagine implementing even something as simple as that in SQL. The fundamental problem is lack of iteration or recursion. (The SQL-99 standard, also known as SQL3, does provide a way to express recursive queries, but many DBMSs don't support it.) This is one reason why we embed SQL in programs written in general-purpose programming languages like Python.

Our goal in Part 3 is for you to get some practise writing psycopg2 code, and to do it in a context that helps you appreciate why and when we might bring in a general-purpose language like Python, without you having to write a lot of Python code.

Getting to know the schema

Read the schema we have provided in `schema.ddl`, which is available on Quercus. You might also find inspecting the sample data provided in `data.sql` helpful.

To get familiar with the schema, ask yourself questions like these (but do not hand in your answers):

- Can a customer make multiple purchases at the same time?
- Can a customer purchase multiple items as part of the same purchase?
- Can two different customers use the same credit card?
- Why doesn't the `LineItem` table have to record the client id of the client that purchased the item?
- Can a customer review an item they did not purchase?
- Can a customer mark their own review as helpful?
- Can a customer's numeric rating of an item be 0?

Part 1: SQL Queries

General requirements

To ensure that your query results match the form expected by the auto-tester (attribute types and order, for instance), We am providing a schema for the result of each query. These can be found in files `q1.sql`, `q2.sql`, . . . , `q6.sql`. You must add your solution code for each query to the corresponding file.

At the beginning of the DDL file defining our database schema, we set the search path to `Recommender`, so that the full name of every table etc. that we define is actually `Recommender.whatever`. We have set the search path to `Recommender` at the top of the query files too, so that you do not have to use the `Recommender` prefix throughout. Do not change this, or any of the provided code.

You are encouraged to use views to make your queries more readable. However, you must **make sure that each file is entirely self-contained, and not depend on any other files**; each will be run separately on a fresh database instance, and so (for example) any views you create in `q1.sql` will not be accessible in `q5.sql`.

The output from your queries must exactly match the specifications in the question, including attribute names, attribute types, and attribute order. Unless otherwise specified, row order does not matter. Your code must work on any database instance (including ones with empty tables) that satisfies the schema.

The queries

These queries are quite complex, and we have tried to specify them precisely. If behaviour is not specified in a particular case, we will not test that case.

Design your queries with the following in mind:

- We use the terms "order" and "purchase" interchangeably.
- A row in the Review table counts as a review even if it has no comment associated with it.
- Do not do any rounding or truncating of values.

Write SQL queries for each of the following:

1. **Unrated products** Customers who buy unrated products are of interest when marketing new products. Find all customers who have bought at least three different items that do not have any reviews, not even a rating without a comment. Report the customer's CID, first name, last name and email.

Attribute	
CID	The customer's CID.
first_name	The customer's first name.
last_name	The customer's last name.
email	The customer's email
Everyone?	Include only customer(s) who meet the criteria.
Duplicates?	No customer should appear twice.

2. **Helpfulness** A review is considered helpful if it has been rated for helpfulness and it has received more helpfulness ratings of True than False. A customer's helpfulness score is a number between 0 and 1, and is defined as their number of helpful reviews divided by the total number of reviews they have written. If a customer has never been rated on helpfulness or has never written a review, then their helpfulness score is zero.

   For each customer, report their CID, name (first name and last name), and a helpfulness category of either:

   - "very helpful" if their score is at least 0.8,
   - "somewhat helpful" if their score is at least 0.5 and less than 0.8, or
   - "not helpful" if their score is less than 0.5.

Attribute	
CID	The customer's CID.
name	The customer's full name as "firstName lastName" e.g., if `firstName` is Diane and `lastName` is Horton, then name is "Diane Horton"
helpfulness_category	The customer's helpfulness category as defined above.
Everyone?	All customers should be included in the result.
Duplicates?	No customer should appear twice.

3. **Curators** Let's say that a customer is a "curator" for a category if they have bought and reviewed every item in that category, and all of those reviews contain a comment i.e., the comment is not `NULL`. Any non-NULL comment counts towards curator status even if the comment is the empty string. For each curator, report their CID and the name of the category for which they are a curator. If a customer is a curator for more than one category, report a row for each.

Attribute	
CID	The customer's CID.
category_name	The name of the category for which the customer is a curator.
Everyone?	Only include customers who are a curator of one or more categories.
Duplicates?	While a customer might appear multiple times (once for each category, for which they are a curator), a customer, category pair should appear at most once in the result.

4. **Best and worst categories** The sales value in a month for a category is the total dollar value of all purchases of items in that category in that month. For each month of the year 2024, find the categories which have had the highest and lowest sales value in that month. Report the month, name of the category with the highest sales, value for the highest category, name of the category with the lowest sales, and value for the lowest category. Include categories that did not have any sales (their sales value is 0). Every month must be in the result, even if it didn't have any sales.

   If there are ties, report them all. Because there can be ties for both highest and lowest, these will "multiply out". For example, if in some month 3 categories are tied for highest sales and there are 2 categories with the lowest sales, there will be 6 rows in the result for that month.

Attribute	
month	A month in 2024 as a string i.e., `'01'`, .`'02'`, ...,
highest_category	The name of the category with the highest sales value.
highest_sales_val	The highest sales value.
lowest_category	The name of the category with the lowest sales value.
lowest_sales_val	The lowest sales value.
Everyone?	Report only categories that satisfy the criteria.
Duplicates?	Each month should appear exactly once in the result. A category could appear multiple times in the result.

5. **Hyperconsumers** Let's define a "hyperconsumer" for a given year to be a customer whose total number of units of all items (total quantity) bought in that year is among the top 5 highest. If there are ties, there could be more than 5 hyperconsumers in a year. For example, if the top consumers for a year, in order, were:

A	926 units
B	901 units
C	901 units
D	884 units
E	850 units
F	790 units
G	790 units
H	790 units
I	722 units

   lower amounts for the rest

   then the 5 highest numbers of units are 926, 901, 884, 850, and 790, and there are 8 hyperconsumers (customers A through H). On the other hand, if few customer made purchases in a year, there may be fewer than 5 hyperconsumers. In the extreme case, in a year where no one bough anything, there would be none.

   For each year in the database, report a row for each hyperconsumer that year. The row should contain the year, name of that hyperconsumer, their email address, and number of units bought. For name, concatenate the first and last name separated by a space.

Attribute	
year	A year recorded in our database, formatted as a string of 4 characters e.g. `'2025'`.
name	A hyperconsumer for the respective year, reported as "firstName lastName" e.g., if `firstName` is Diane and `lastName` is Horton, then name is "Diane Horton"
email	The email of the hyperconsumer.
items	The number of units this hyperconsumer bought this year.
Everyone?	Only report hyperconsumers.
Duplicates?	While a customer could be a hyperconsumer during multiple years. Only report the year, customer combination once.

6. **Year-over-year sales** We will define the "operational years" as the years that fall in the range between the earliest year with a purchase in the database to the most recent year with a purchase in our database. The total unit sales for an item in a given month of a year is the total quantity purchased of that item during that month of that year (which could be zero). The average unit sales for an item in a given year is the average of the item's monthly total unit sales over the 12 months of that year. For each item in our database (even if it

was never purchased), and for each pair of consecutive "operational years", report the year-over-year change in the item's average unit sales, which can be calculated as:

$$\left(\frac{\text{year2 average} - \text{year1 average}}{\text{year1 average}}\right) \times 100$$

This is the percentage decrease or increase from one year to the next. For example, if an item had an average unit sale of 100 units in 2023 and 125 units in 2024, its year-over-year change was 25%. If the average for both year 1 and year 2 is 0 then report 0. If the average for year 1 is 0 but year 2's average is greater than 0, report '`Infinity`' (this is a special value you can report in a `float` column).

There must be no gaps in the pairs of consecutive years reported, even if some years had no purchases. For example, if the database includes purchases made in 2020, 2022 and 2025, the "operational" years are all years in the range [2020, 2025]. Your result should include information about pairs of consecutive years where both years are in that range i.e., $\langle 2020, 2021\rangle, \langle 2021, 2022\rangle, ..., \langle 2024, 2025\rangle$.

Your result should report the item id, the first year in a pair of years, the average for that year, the second year in a pair of years, the average for the second year, and the change as a FLOAT.

Attribute	
IID	The item ID of an item in our database.
year1	The first year in the pair of consecutive operational years as an integer.
year1_avg	The average sales in year 1.
year2	The second year in the pair of consecutive operational years as an integer.
year2_avg	The average sales in year 2.
yoy_change	The year-over-year change between year1 and year2 as described above.
Everyone?	All items in the database, as well as all pairs of consecutive operational years must be reported.
Duplicates?	An item or a year might be reported multiple times, but the combination ⟨ IID, year1, year2⟩ must appear at most once.

SQL Tips

- There are many details of the SQL library functions that we are not covering. It's just too vast! Expect to use the postgreSQL documentation to find the things you need. Chapter 9 on functions and operators is particularly useful. Google search is great too, but the best answers tend to come from the official postgreSQL documentation.
- You may use any features defined in our version of PostgreSQL on the Teaching Labs. This is not a pointed hint; we have not deliberately left features for you to discover that will dramatically simplify your work. However, we do expect that you will need to look up details in the documentation, and in fact being able to do that quickly and effectively is one of the learning outcomes of the assignment.
- When dealing with a value of type `TIMESTAMP`, `DATE`, or `TIME`, the `EXTRACT` function is handy for pulling out pieces. You might also find the `date_trunc` function helpful. It truncates a timestamp to the specified precision. You might also find the `to_char` function helpful for converting a timestamp to a string according to a specified format.
- You can subtract an interval from a timestamp value. For example:

```
csc343h-marinat=> SELECT NOW() - INTERVAL '1:15:00' AS some_time_ago;
          some_time_ago
-------------------------------
 2026-02-14 23:41:21.072811-05
(1 row)
```

  The above expression yields a timestamp that is 1 hour and 15 minutes before the current time.

- You might find using a `LIMIT` clause helpful.

- You might find the following helpful to make your solutions more succinct.:
  - `COALESCE(value [, ...])` to return the first of its arguments that is not null.
  - A `CASE` expression to generate a different value based on a condition:

    ```
    CASE WHEN condition THEN result
    [WHEN ...]
    [ELSE result]
    END
    ```

  The PostgreSQL documentation cover these in more details.
- You may find this code helpful. It creates the 12 months as text in the format `'MM'`.

  ```
  CREATE VIEW Months AS
  SELECT to_char(generate_series(1, 12), 'FM09') AS mo;
  ```

- This code creates a table with a series of values. It doesn't have a FROM because it doesn't need to refer to any tables to do its job.

  ```
  csc343h-dianeh=> select generate_series(3, 8) as range;
   range
  -------
       3
       4
       5
       6
       7
       8
  (6 rows)
  ```

- Please use line breaks so that your queries do not exceed an 80-character line length.

Part 2: SQL Updates

General requirements

In this section, you will write SQL statements to perform updates. Some questions can be accomplished a single DELETE, UPDATE, or INSERT statement, but others will require several steps.

The updates

Write SQL code for each of the following:

1. **SALE!SALE!SALE!** It's time for the annual site-wide mega sale and item prices need to be set to the new sale prices. For any item that has sold a total of at least ten units, you will need to set the item price by applying a discount as follows:
   - If the item costs between $10 and $50 inclusive, then apply a 20% discount;
   - if the item costs more than $50 but at most $100, then apply a 30% discount; and
   - if the item costs more than $100, apply a 50% discount.
   - There is no discount on items under $10.

2. **Fraud Prevention** In order to prevent credit card fraud, the online store has a limit on the number of purchases that can be made using the same credit card in 24 hours. Find any credit card that has been used on more than five purchases in the last 24 hours. Delete any purchases made after the fifth order, as well as all line items for those purchases. You can use `NOW()` to get the current date and time.

   Don't worry about the exact edge of those 24 hours (whether to use $<$ or $\leq$ and so on). We won't test your code on cases where that makes a difference.

3. **Customer appreciation week** This week, the store wants to reward all customers who make a purchase with a free gift. Customers who placed an order any time yesterday (the promotion period) will automatically receive a free item.

   Start by adding a record for the free item. The item is a mug with description "Company logo mug". You can assume that this item does not currently exist in the `Item` table. The category for this item is "Housewares" and the price is free. The item's ID should be 1+ the maximum item ID currently recorded in our table or 1 if our `Item` table is empty.

   Next, find all customers who placed an order yesterday and add a line item for the free item with a quantity of one to the first order that each customer placed yesterday. (They may have made multiple separate purchases yesterday.)

   Don't assume that purchase ID's are in the same order as purchase dates.

Part 3: Embedded SQL with psycopg2

For Part 3, you will complete methods in **a2.py**. The code provides infrastructure for a recommender system. It includes two different methods that can recommend items for purchase: **`recommend_generic`** makes recommendations 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.

Both recommendation methods depend on identifying highly-rated popular items. Additionally, the **`recommend`** method compares a customer's taste to those of customers whose taste we value. We will refer to these customers as "elite customers". The **`recommend`** method needs to identify those "elite customers"' as well as their ratings of highly-rated popular items, which we will dub "elite ratings".

As you might imagine, identifying these highly-rated popular items as well as "elite customers" and their "elite ratings" is quite costly. If you consider that recommendations will be given very often, this approach doesn't seem practical. However, do we need to recalculate this information 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 "elite 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 "elite ratings" at most an hour ago. So we can save a "snapshot" of the "elite ratings" periodically and make our recommendations based on those. Having to compute the "elite ratings" only periodically rather than once per recommendation would certainly save a lot of computation!

The snapshot for our highly-rated popular items, "elite customers" and "elite ratings" will reside in the `PopularItem`, `EliteMember` and `EliteRating` tables respectively. The **`repopulate`** method is responsible for [re-]building the "snapshot", 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. To simplify your task, you won't be required to populate the `EliteMember` table.

Your task

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

1. `repopulate`: Updates the "snapshot" of data in tables .
2. `recommend_generic`: Recommends items to the customer based on item ratings.
3. `recommend`: Recommends items to the customer based on the ratings of the most similar "elite customer".

You will have to decide how much to do in SQL and how much to do in Python. You could use the database for very little other than storage: for each table, you could write a simple query to dump its contents into a data structure in Python and then do all the real work in Python. This is a bad idea. The DBMS was designed to be extremely good at operating on tables! You should use SQL to do as much as it can do for you.

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

Important Guidelines

- Feel free to refer to the psycopg2 documentation: `https://www.psycopg.org/docs/`.
- Do not use standard input in the methods you are completing (`repopulate`, `recommend_generic`, and `recommend`) in `a2.py`, and any helper methods they call. Doing so will result in the autotester timing out, causing you to receive a **zero** on that method. However, you can use standard input in any testing code that you write outside of these methods, including the main block and testing functions.
- We have included a basic test suite in `test_preliminary.py`. This suite is not comprehensive and is intended as an example of how you might test your code. You are responsible for thoroughly testing your implementation to ensure its correctness. You can change this file freely and you should not submit it.
- Do not change any of the code provided `a2.py`. In particular, you may not change the header of any of the methods we've asked you to implement. Each method must have a try-except clause so that it cannot possibly throw an exception.
- You have been provided with methods called `connect()` and `disconnect()` that allow you to respectively connect to and disconnect from the database. You must **NOT** make any modifications to either method.
- Do not hardcode your connection information anywhere in the `Recommender` class. Our autotester will use the `connect()` and `disconnect()` methods to connect to the database with our own credentials.
- You should **NOT** call `connect()` and `disconnect()` 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. Alternatively, you can declare your view as temporary so that it is dropped automatically once the connection is closed. The syntax for this is `CREATE TEMPORARY VIEW name AS ...`
- 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). Don't do those extra things.
- If behaviour is not specified in a particular case, we will not test that case.
- Do not write any code outside of a function/method or the main block.

Some Python tips

Some of your SQL queries may be very long strings. You should write them on multiple lines, both for readability and to keep your code within an 80-character line length. You can achieve that by using Multi-line strings in Python. Multi-line strings are declared using triple quotes, and are allowed to span multiple lines.

```
sql_query = """
    SELECT first_name, last_name
    FROM Customer
    WHERE title <> 'Customer'
"""
```

Alternatively, You can break the string into pieces and use + to concatenate them together. Don't forget to put a blank at the end of each piece so that when they are concatenated you will have valid SQL. Example:

```
sql_query =
   "SELECT first_name, last_name " +
   "FROM Customer " +
   "WHERE title <> 'Customer'";
```

How your work 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), we will provide a checker that you can run through MarkUs. If the checker passes, this tells you only that your code runs on our system.

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. Your work will be marked only for correctness.

Some advice on testing your code

Testing is a significant task, and is part of your work for A2. You will need a dataset for each condition / scenario you want to test. These can be small, and they can be minor variations on each other.

We 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.sql`. 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 your embedded SQL code can be done in a similar way to the "Coordinating" part of the Embedded exercises posted on our Lectures page. We 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. Alternatively, you can skip this step and instead provide the path to the schema and dataset files to the `setup` function in the `test_preliminary.py` file.
3. In window 2 (remember, this is on dbsrv1), make the necessary modifications to run the method you want to test. You can modify the main block of your a2 program so that it has an appropriate call to the method. Alternatively, you can use a test function like the ones we have provided in `test_preliminary.py`.
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. The comments in `test_preliminary.py` provide more tips on how to test your code. 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.

Submission instructions

You must declare your team on MarkUs even if you are working solo, and must do so before the due date. If you plan to work with a partner, declare this as soon as you begin working together. If you need to dissolve your group, you need to contact us.

For this assignment, you will hand in numerous files. MarkUs shows you if an expected file has not been submitted; check that feedback so you don't accidentally overlook a file. Also check that you have submitted the correct version of your file by downloading it from MarkUs. New files will not be accepted after the due date.

This assignment will be autotested. It is your responsibility to make sure your filenames and contents are what you expect. There are no remarks on autotested assignments.