University of Toronto
CSC343, Fall 2022

Assignment 2

Due: Wed, November 2, before 4pm

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 documentation
- embed SQL in a high-level language using psycopg2 and Python
- recognize limits to the expressive power of standard SQL

Please read this assignment thoroughly before you proceed.

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

In this assignment, we will work with a database to support a ride-sharing / taxi company like Uber or Lyft. Keep in mind that your code for this assignment must work on *any* database instance (including ones with empty tables) that satisfies the schema.

Part 1: SQL Queries

General requirements

To ensure that your query results match the form expected by the autotester (attribute types and order, for instance), We are providing a schema for the result of each query. These can be found in files `q1.sql`, `q2.sql`, ..., `q10.sql`. You must add your solution code for each query to the corresponding file. Make sure that each file is entirely self-contained, and does 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 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:

- When we say that a client *had a ride*, or a driver *gave a ride* we mean that the ride was completed, that is, it has gone from request through to drop-off.
- The date of a ride is the date on which it was requested. (The drop-off might have a later date if the ride began just before midnight, for example.) Similarly, the month of a ride and the year of a ride are determined by the date on which it was requested.
- When we refer to a month we mean a specific month and year combination, such as January 2022, rather than just January.
- We will assume the following constraints hold:
  - The end time of a shift is after its start time.
  - A shift will have at least one row in table Location, recorded at the start of the shift, and it may have more. Each additional row for a shift indicates an updated location for the driver, and will have a datetime that occurs between the shift start time and the shift end time (inclusive).
  - The request, dispatch, pickup and dropoff for any given ride occur in that order in time, and each step is after (not at the same time) as the one before.
  - No dispatch can be recorded for a driver while they have another ride that has not been completed.
  - No ride request can be recorded for a client if they have another ride request that has not lead to a completed ride.

  If it weren't costly to enforce these restrictions, we would express them as constraints.

Write SQL queries for each of the following:

1. **Months.** For each client, report their client ID, email address, and the number of different months in which they have had a ride. January 2021 and January 2022, for example, would count as two different months.

Attribute	
client_id	id of a client
email	email address of this client
months	the number of different months in which they have had a ride.
Everyone?	Every client should be included, even if they have never had a ride.
Duplicates?	No client can be included more than once.

2. **Lure them back.** The company wants to lure back clients who formerly spent a lot on rides, but whose ridership has been diminishing.

   Find clients who had rides before 2020 costing at least $500 in total, have had between 1 and 10 rides (inclusive) in 2020, and have had fewer rides in 2021 than in 2020.

Attribute	
client_id	id of a client who meets the criteria of this question.
name	The client's first name and surname name, with a single blank between. Do not add any punctuation, but if either attribute contains punctuation, leave it as is.
email	email address of this client. If an email address is NULL, report it as the string "unknown".
billed	total amount the client was billed for rides that occurred prior to 2020.
decline	difference between the number of rides they had in 2020 and the number in 2021.
Everyone?	Include only clients who meet the criteria of this question.
Duplicates?	No client can be included more than once.

3. **Rest bylaw.** A *break* is the time elapsed between one drop-off by a driver and their next pick-up on that same day (even if the pickup of the first ride was on a different day). The *duration* of a ride is the time elapsed between pick-up and drop-off (If a ride has a pick-up time recorded but no drop-off time, it is incomplete and does not have a duration). The *total ride duration* of a driver for a day is the sum of all ride durations of that driver for rides whose pickup and drop-off are both recorded and were both on that day.

   We will treat a day as going from midnight to midnight.

   A city bylaw says that no driver may have three consecutive days where on each of these days they had a total ride duration of 12 hours or more yet never had a break lasting more than 15 minutes. Keep in mind that a driver could have a day with a single ride and nothing that counts as a break. They would by definition violate the bylaw on that day if the ride was long enough.

   Find every driver who broke the bylaw. Report their driver ID, the date on the first of the three days when they broke the bylaw, their total ride duration summed over the three days, and their total break time summed over the three days.

   If a driver has broken the bylaw on more than one occasion, report one row for each. Don't eliminate overlapping three-day stretches. For example, if a driver had four long workdays in a row, they may have broken the bylaw on the sequence of days d1, d2 and d3, and also on the sequence of days d2, d3, and d4. There would be two rows in your table to describe this.

   Your query should return an empty table if no driver ever broke the bylaw.

Attribute	
driver_id	driver ID of a driver who broke the bylaw
start	date on the first in a sequence of three days when they broke the bylaw
driving	their total ride duration on the three days, in hours, minutes and seconds, *e.g.,* `13:24:54`
breaks	their total break time summed over the three days, again in hours, minutes and seconds
Everyone?	Include only drivers who meet the criteria of this question.
Duplicates?	A driver appears once per violation of the bylaw, but no entire row will be a duplicate.

4. **Do drivers improve?** The company offers optional training to new drivers during their first 5 days on the job, and wants to know whether it helps, or whether drivers get better on their own with experience.

   A driver's first day on the job is the first day on which they gave a ride. Consider those drivers who have had the training and have given a ride (one or more) on at least 10 different days. Let's define their *early average* to be their average rating in their first 5 days on the job, and their *late average* to be their average rating after their first 5 days on the job. Report the number of such drivers, the average of their early averages, and the average of their late averages. Do the same for those drivers who have *not* had the training but have given a ride (one or more) on at least 10 different days.

   A driver's first 5 days on the job are the first 5 days on which they gave a ride. These need not be consecutive days.

   NULL values should not contribute to an average. The `average` function in SQL takes care of this for you.

   A driver's early average is NULL if none of the rides in their first 5 days were rated. Their late average is NULL if none of the rides they have given after their first 5 work days were rated.

Attribute	
type	either 'trained' or 'untrained'
number	the number of drivers of that type (trained or untrained) who have given a ride (one or more) on at least 10 different days
early	the average early average of such drivers, or NULL if all of their early averages are NULL
late	the average late average of such drivers, or NULL if all of their late averages are NULL
Everyone?	Only drivers who have given a ride (one or more) on at least 10 different days are included in the statistics.
Duplicates?	No. There will be just two rows, one for 'trained' and one for 'untrained'.

5. **Bigger and smaller spenders.** For each client, and for each month in which someone had a ride (whether or not this client had any rides in that month), report the total amount the client was billed for rides they had that month and whether their total was at or above the average for that month or was below average. The average for a month is defined to be the average total for all clients who completed at least one ride in that month.

Attribute	
client_id	client ID
month	a month in which someone had a ride, as a 7-character string in this format: '`2021 07`'
total	the total amount this client was billed for rides they had that month.
comparison	either 'below' or 'at or above'
Everyone?	Include every combination of a client and a month in which someone (not necessarily that client) had a ride.
Duplicates?	There can be no duplicates.

6. **Frequent riders.** Find the 3 clients with the greatest number of rides in a single year and the 3 clients with the smallest number of rides in a single year. Consider only years in which some client had a ride.

   There may be ties in number of rides. You should include *all* clients with the highest number of rides, all clients with the second highest number of rides, and all client with the third highest number of rides. Do the same for clients with the lowest 3 values for number of rides. As a result, your answer may actually have more than 6 rows. A single client could appear more than once with the same number of rides, if they had that number of rides in two different years and that number was among the top or bottom 3, or both, but don't repeat the same client-year-rides combination.

Attribute	
client_id	client ID for a client who had a top-3 year or a bottom-3 year
year	a year when this client's number of rides was in the top 3 or bottom 3, as a 4-character string in this format: '`2020`'
rides	the number of rides this client had in this year.
Everyone?	Include only top-3 and bottom-3 results. There may be more than 6 rows in total.
Duplicates?	There can be no duplicates.

7. **Ratings histogram.** We need to know how well-rated each driver is. Create a table that is, essentially, a histogram of driver ratings.

Attribute	
driver_id	id of the driver
r5	Number of times they received a rating of 5.
r4	Number of times they received a rating of 4.
r3	Number of times they received a rating of 3.
r2	Number of times they received a rating of 2.
r1	Number of times they received a rating of 1.
Everyone?	Every driver should be included, even if they have no ratings.
Duplicates?	No driver can be included more than once.

8. **Scratching backs?** We want to know how the ratings that a client gives compare to the ratings that the same client gets. Let's say there is a reciprocal rating for a ride if both the driver rated the client for that ride and the client rated the driver for that ride.

   For each client who has at least one reciprocal rating, report the number of reciprocal ratings they have, and average difference between their rating of the driver and the driver's rating of them for a ride.

Attribute	
client_id	ID of a client who has at least one reciprocal rating
reciprocals	number of reciprocal ratings they have
difference	average difference between their rating of the driver and the driver's rating of them; a positive value indicates that they rate drivers higher than drivers rate them, on average; a negative value indicates that they rate drivers lower than drivers rate them, on average
Everyone?	Include only clients who have at least one reciprocal rating.
Duplicates?	There can be no duplicates.

9. **Consistent raters.** Report the client ID and email address of every client who has rated every driver they have ever had a ride with. (They needn't have rated every ride with that driver.) Don't include clients who have never had a ride.

Attribute	
client_id	ID of a client who has rated every driver they have ever had a ride with.
email	email address of the client, or NULL if there is none recorded
Everyone?	Include only clients who have had a ride and have rated every driver they have ever had a ride with.
Duplicates?	There can be no duplicates.

10. **Rainmakers.** The company wants to know which drivers are earning a lot for the company, and how this has changed over time.

    The *crow-flies distance* of a ride is the number of miles between the source and the destination given in the ride request, "as the crow flies", that is, without concern given to where the streets are. You can compute the distance between two points using the operator `<@>`, as described in `distance-example.txt`. A driver's *total crow-flies mileage* for a month is the total crow-flies distance of rides that they gave in that month. A driver's *total billings* for a month is the total amount billed for rides they gave in that month.

    For every driver, report (a) their total crow-flies mileage and total billings per month for completed billed rides for each month in 2020, (b) the same information for 2021, and (c) the differences between the corresponding months in the two years.

Attribute	
driver_id	id of driver
month	the number of the month to be compared across 2020 and 2021, as a 2-character string, for example '01' for January and '11' for November
mileage_2020	the driver's total crow-flies mileage for this month in 2020
billings_2020	the driver's total billings for this month in 2020
mileage_2021	the driver's total crow-flies mileage for this month in 2021
billings_2021	the driver's total billings for this month in 2021
mileage_increase	the difference between their total crow-flies mileage for this month in 2021 and their total crow-flies mileage for this month in 2020; a positive value indicates a year-over-year increase in total crow-flies mileage.
billings_increase	the difference between their total billings for this month in 2021 and their total fbillings or this month in 2020; a positive value indicates a year-over-year increase in total billings
Everyone?	Every driver should be included, even if they have zero for all values.
Duplicates?	Every driver appears in 12 rows, one for each month.

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.

- When subtracting timestamp values, you get a value of type INTERVAL. If you want to compare to a constant time interval, you can do this, for example:

```
WHERE (a - b) > INTERVAL '0'
```

- 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), '09') AS mo;
```

- 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.

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

Part 2: Embedded SQL

Part 2 will be made available after you have learned about embedded SQL.

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 have provided some self-tests that you can run through MarkUs.

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 queries for Part 1

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. Suggestion: Use names that indicate something about the scenario, like "Big Spender" as a client who has large total billings.

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-drivers-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.

Submission instructions

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

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.