University of Toronto
CSC343, Fall 2019

Assignment 2

Due: Wed, November 13, before 8pm

Learning Goals

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 JDBC
- 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**.

The Domain

In this assignment, we will work with a database to support a ride-sharing / taxi company like Uber. You've already become familiar with the schema through your work for Prep8. 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 Statements

General requirements

In this section, you will write SQL statements to perform queries.

To ensure that your query results match the form expected by the autotester (attribute types and order, for instance), I am 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 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 I 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 date of the drop-off might be later if the ride began just before midnight, for example.)
- When we refer to a month we mean a specific month and year combination, such as January 2016, rather than just January.
- We will assume that the request, dispatch, pickup and dropoff for any give ride occur in that order in time. For example, the dropoff will not occur before the pickup. 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 2018 and January 2019, 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 2014 costing at least $500 in total, have had between 1 and 10 rides in 2014, and have had fewer rides in 2015 than in 2014.

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 "unknown".
billed	total amount the client was billed for rides that occurred prior to 2014.
decline	difference between the number of rides they had in 2014 and the number in 2015.
Everyone?	Include only clients who meet the criteria of this question.
Duplicates?	No client can be included more than once.

3. **Rest bylaw.** 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 were both on that day. A *break* is the time elapsed between one drop-off by a driver and his or her next pick-up on that same day. There can be no break before the first pick-up of the day or after the last drop-off of the day. We will treat a day as going from midnight to midnight.

   A city bylaw says that no driver may have three days in a row 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 therefore no breaks). They would by definition break the rule 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 days d1, d2 and d3, and also on days d2, d3, and d4. There will 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	driver ID of a driver who broke the bylaw
start	date on the first 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 late average is NULL if they have given no rides after their first 5 days on the job. Their early average cannot be NULL because they have given at least one ride in their first 5 days, by definition. NULL values should not contribute to an average.

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
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 an 8-character string in this format: '`2015 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: '`2015`'
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, or null if they never did.
r4	Number of times they received a rating of 4, or null if they never did.
r3	Number of times they received a rating of 3, or null if they never did.
r2	Number of times they received a rating of 2, or null if they never did.
r1	Number of times they received a rating of 1, or null if they never did.
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 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 he or she gave in that month. A driver's *total billings* for a month is the total amount billed for rides he or she gave in that month.

For every driver, report (a) their total crow-flies mileage and total billings per month, for each month in 2014, (b) the same information for 2015, 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 2014 and 2015, as a 2-character string, for example '01' for January and '11' for November
mileage_2014	the driver's total crow-flies mileage for this month in 2014
billings_2014	the driver's total billings for this month in 2014
mileage_2015	the driver's total crow-flies mileage for this month in 2015
billings_2015	the driver's total billings for this month in 2015
billings_increase	the difference between their total billings for this month in 2015 and their total fbillings or this month in 2014; a positive value indicates a year-over-year increase in total billings
mileage_increase	the difference between their total crow-flies mileage for this month in 2015 and their total crow-flies mileage for this month in 2014; a positive value indicates a year-over-year increase in total crow-flies mileage.
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 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 of 2014.

```
CREATE VIEW Months as
SELECT to_char(DATE '2014-01-01' + (interval '1' month * generate_series(0,11)), 'MM') as mo;
```

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

Part 2: Embedded SQL

Imagine an Uber app that drivers, passengers and dispatchers log in to. The different kinds of user have different features available. The app has a graphical user-interface, written in Java, but ultimately it has to connect to the database where the core data is stored. Some of the features will be implemented by Java methods that are merely a wrapper around a SQL query, allowing input to come from gestures the user makes on the app, like button clicks, and output to go to the screen via the graphical user-interface. Other app features will include computation that can't be done, or can't be done conveniently, in SQL.

For Part 2 of this assignment, you will not build a user-interface, but will write several methods that the app would need. It would need many more, but we'll restrict ourselves to just enough to give you practise with JDBC and to demonstrate the need to get Java involved, not only because it can provide a nicer user-interface than postgreSQL, but because of the expressive power of Java.

General requirements

- You may not use standard input in the methods that you are completing. Doing so will result in the autotester 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.)
- You may not change the header of any of the methods we've asked you to implement, not even to declare that a method may throw an exception. Each method must have a try-catch clause so that it cannot possibly throw an exception.
- You will be writing a method called `connectDb()` to connect to the database. When it calls the `getConnection()` method, it must use the database URL, username, and password that were passed as parameters to `connectDb()`; these values must not be "hard-coded" in the method. Our autotester will use the `connectDb()` and `disconnectDB()` methods to connect to the database with our own credentials.
- You should **not** call `connectDb()` and `disconnectDB()` 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 `Assignment2.java`. This is the only file you may submit for this part.
- You are welcome to write helper methods to maintain good code quality.
- Do only what the Javadoc 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.

Your task

Complete the following methods in the starter code in `Assignment2.java`:

1. `connectDB`: Connect to a database with the supplied credentials.
2. `disconnectDB`: Disconnect from the database.
3. `available`: A method that would be called when the driver declares his or her availability to pick up a client.
4. `pickedUp`: A method that would be called when the driver declares that he or she has picked up a client.
5. `dispatch`: A method that would be called when the dispatcher chooses to dispatch drivers to pick up those clients who've requested rides within a geographical area.

You will have to decide how much to do in SQL and how much to do in Java. At one extreme, 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 Java and then do all the real work in Java. 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. In particular, there is no need to use a Java data structure such as an ArrayList, HashMap, or Set, or even a simple array.

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

Using values of type point

Some of the columns in our tables have type `point`. In Part 2 of the assignment, you'll work with these values. In SQL, here's how to access the x and y coordinates within a point:

> "It is possible to access the two component numbers of a point as though the point were an array with indexes 0 and 1. For example, if t.p is a point column then `SELECT p[0] FROM t` retrieves the `X` coordinate and `UPDATE t SET p[1] = ...` changes the `Y` coordinate."

(Reference: `http://www.postgresql.org/docs/9.4/static/functions-geometry.html`)

When you need to store a point in your JDBC code, you will use a Java object of type `PGpoint`. We have provided the necessary import for this in the starter code. In order to compile and run your JDBC code with this import, you will need to provide a different class path than we have been using. And you'll have to use it for the compile step also. Here are the exact commands to use:

```
javac -cp /local/packages/jdbc-postgresql/postgresql-42.2.4.jar Assignment2.java
java -cp /local/packages/jdbc-postgresql/postgresql-42.2.4.jar: Assignment2
```

Notice that there is a colon in the run step but not the compile step.

Normally, when you want to grab an attribute from a row of a result set, you call the appropriate method for that type of value. For example, to retrieve an `int` attribute, you can use method `rs.getInt()`. Similarly, when you want to set a value in a prepared statement, you might call method `ps.setInt()`. There are no analogous methods for values of type `PGpoint`. Instead we use the methods designed for values of the generic Java type `Object`. For example:

```
// Get the value of the third attribute in this row of the result set.
// It comes out as an Object, then we cast it as a PGpoint.
PGpoint sourceLocation = (PGpoint) rs.getObject(3);

// Set the second ? value in this prepared statement to sourceLocation.
// setObject will accept any kind of Object, including a PGpoint.
ps.setObject(2, sourceLocation);
```

Additional JDBC tips

Some of your SQL queries may be very long strings. You should write them on multiple lines for readability, and to keep your code within an 80-character line length. But you can't split a Java string over multiple lines. You'll need to 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:

```
String sqlText =
   "select client_id " +
   "from Request r join Billed b on r.request_id = b.request_id " +
   "where amount > 50";
```

Please refer to the JDBC exercise from class for some common mistakes and the error messages they generate.