University of Toronto
CSC343, Fall 2016

Assignment 2

Due: Wednesday, November 16, at 8:00 pm sharp!

Learning Goals

By the end of this assignment you should have:

1. become fluent in SQL
2. learned your way around the postgreSQL documentation
3. learned how to embed SQL in a high-level language using JDBC
4. encountered limits of the expressive power of standard SQL

General Instructions

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

We strongly encourage you to do all your development for this assignment on the CS Teaching Labs, either in the lab rooms or via a remote connection. fYour code must run on these machines in order to earn credit.

Download the starter files from the course webpage:

- The database schema, `bnbSchema.ddl`
- A very small sample data set (with just enough data to show you how to format your own), `bnbData.sql`

If you are using a graphical user interface to connect to the CS Teaching Labs, you can simply download the files and save them into a directory of your choice. You can also use the command line to download the files. For example:

```
> wget http://www.teach.cs.toronto.edu/~csc343h/fall/assignments/a2/bnbSchema.ddl
```

You are allowed, and in fact encouraged, to work with a partner for this assignment. You must declare your team (whether it is a team of one or of two students) and hand in your work electronically using MarkUs.

Once you have submitted your files, be sure to check that you have submitted the correct version; new or missing files will not be accepted after the due date, unless your group has grace tokens remaining.

Schema

In this assignment, we will work with a database to support a home-sharing company like Airbnb.

Download the schema and small sample dataset. Your code for this assignment must work on *any* database instance (including ones with empty tables) that satisfies the schema, so make sure you read and understand them.

The schema definition uses several types of integrity constraints:

- Every table has a primary key ("`PRIMARY KEY`").
- Some tables use foreign key constraints ("`REFERENCES`"). By default, they will refer to the primary key of the other table.
- Some attributes must be present ("`NOT NULL`").

Warmup: Getting to know the schema

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

- What does the database look like when a traveler has requested a booking, but has not yet been accepted? How does it evolve between the booking request and (for successful requests) the end of the stay when the traveler has checked out?
- How is the cost of a booking recorded? Can a booking have a cost recorded before the stay is complete?
- Can a traveler rate the same owner more than once? Can an owner rate a traveler more than once?

Part 1: SQL Statements

In this section, you will write SQL statements to perform queries. Write your SQL statement(s) for each question in separate files named `q1.sql`, `q2.sql`, ..., `q8.sql`, and submit each file on MarkUs. You are encouraged to use views to make your queries more readable. However, each file should be 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`. **Each of your files must begin with the line** `SET search_path TO bnb, public;` Failure to do so will cause your query to raise an error, leading you to get a 0 for that question.

The output from your queries must exactly match the specifications in the question, including attribute names and order, and the order of the tuples.

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**.

Write SQL queries for each of the following:

1. **Traveler Statistics** For each of the last 10 years, and for each traveler, report their ID, email address, the number of booking requests made and the number of bookings. Consider a request and booking to be made in a year if the startdate is within that year.

Attribute	
travelerId	id of a traveler
email	email address of this traveler
year	the year of request or booking
numRequests	number of requests
numBooking	number of bookings
Order by	year descending
Everyone?	Every traveler should be included, even if they have never made a request or booking
Duplicates?	No traveler can be included more than once per year

2. **Find Scrapers** A scraper is someone who puts in a lot of booking requests in order to *scrape* the information about properties without ever booking a stay.

   Find travelers who have placed more than 10 times the average number of booking requests (averaged across all travelers) without ever making a booking. Report the traveler's personal information along with his or her most requested city. Also report the total number of booking requests for the traveler. If there are ties for the most requested city, report the city that comes first alphabetically among all the ties.

   When reporting a traveler's name, create a string that combines the customer's first name and surname, with a single blank between. Do not add any punctuation, but if either attribute contains punctuation, leave it as is. For this query, if an email address is NULL, report it as "unknown".

Attribute	
travelerId	id of a traveler who meets the criteria of this question.
name	Name of the traveler
email	email address of this traveler, or "unknown"
mostRequestedCity	city that was the most requested by this traveler
numRequests	total number of booking requests by this traveler
Order by	the total number of booking requests, descending and then by travelerID, ascending
Everyone?	Include only travelers who meet the criteria of this question.
Duplicates?	No traveler can be included more than once.

3. **Short-term rental bylaws.**

   Cities often impose a *room tax* on hotels. To protect the hotel industry (and city coffers), some cities have bylaws that restrict how private property can be rented. For example, in New York City the law bars apartment-dwellers from renting out their units for less than 30 days (meaning a single booking cannot be less than 30 days). In Berlin, no house can be rented for less than two months. Both of these bylaws specify a *minimum* number of days per rental. Other cities specify a *maximum* number of days per year that a property can be rented. For example, San Francisco limits rentals (on all property types) to a maximum of 90 days per year. The maximum limits are not on a single rental, but rather on the total number of days rented over all bookings in a year. Not all cities have short-term rental bylaws. But a city can have both a maximum and a minimum bylaw. The property type for a bylaw can be any of the BnB property types. If the property type is null, that means the bylaw applies to all property types.

   Write a query to return the homeowner (id, first and surnames), and listingId and year for all listings that have violated any of the bylaws in the city of the listing. When you check a bylaw that sets a maximum number of days per year, you may encounter bookings that span two years. In that case, you will need to add up the number of nights for that booking in each of the years. For example, a booking starting Dec 30th lasting 5 nights contains two nights booked in 2015 (Dec 30th and 31st) and three nights booked in 2016 (Jan 1st through 3rd).

   Notice that the schema ensures each listing has at most one booking that starts on any given date. But it does not ensure that bookings for a single listing are non-overlapping. So for this question, you should first determine if the bookings for a listing are *valid*, meaning non-overlapping. For example, if there is a 5 night booking for listing 1 starting Jan 1, 2016 and another booking starting Jan 3, 2016 for the same listing, then the bookings for listing 1 are invalid. You should only check for violations among listings that are valid.

   Your query should return an empty table if no violations occur in the database among listings with valid bookings.

Attribute	
homeowner	homeownerID who broke the bylaw
listingID	listingID of the property that broke the bylaw
year	year when it broke the bylaw
city	city of the property
Everyone?	Include only homeowners/properties who meet the criteria of this question.
Duplicates?	A homeowner/property appears at most once per year (even if they violate both a maximum and minimum bylaw)

4. **Do Homeowners improve?** Homeowner ratings are recorded for a specific booking that happened on a specific date. BnB would like to understand if their homeowners are taking their ratings seriously and improving their ratings over time. For this question, you will need to compute a homeowner's average rating over a year. Please consider a rating to be in Year Y, if the `startdate` of the `booking` for which the rating is recorded is in Year Y.

   Write a query to return the percentage of homeowners whose average ratings are *monotonically non-decreasing* (meaning that for Year Y and Year W, if Y $<$ W, then average rating in Year Y is $\leq$ average rating in Year W), over the last ten years.

Here's how to handle some important corner cases:

- A homeowner contributes to the percentage iff he or she has 1 or more ratings over the last 10 years.
- If a homeowner has 1 or more ratings over the last 10 years, but not in every one of those years, then:
  - If, in the years when they do have an average rating, the average ratings are monotonically non-decreasing, they satisfy the query.
  - If, in the years when they do have an average rating, the average ratings are **not** non-decreasing, then they do not satisfy the query.

For example, consider the following case.

H1 has an average rating in every one of the last 10 years and they are non-decreasing.
H2 has an average rating in every one of the last 10 years and they are not non-decreasing.
H3 has no ratings in the last 10 years.
H4 only has ratings in the last year (and therefore only a single average).
H5 has ratings last year (averaging 4.5) and the year before (averaging 4) but no other ratings.
H6 has ratings 10 years ago averaging 4, and 7 years ago averaging 3.

The percentage you would report would be 60%. There are 3 homeowners who satisfy the query (H1, H4 and H5) out of a total of five homeowners who contribute to the percentage (H1, H2, H4, H5, H6). Homeowner H3 does not contribute to the percentage. Homeowner H4 satisfies the query trivially.

Attribute	
percentage	percentage of homeowners with non-decreasing average ratings in last 10 years
Everyone?	Only homeowners who have at least one rating in last 10 years
Duplicates?	No. There will be at most one row. No rows if there are no homeowners who have at least one rating in last 10 years.

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

Attribute	
homeownerID	id of the homeowner
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.
Order by	number of 5-ratings descending, then 4-ratings, and so on; if there are still ties, then by homeownerID ascending.
Everyone?	Every homeowner should be included, even if they have no ratings.
Duplicates?	No homeowner can be included more than once.

6. **Committed Travelers.**

   A committed traveler is one who has booked every listing on which they have made a booking request. A traveler may have put in several booking requests on the same listing but as long as he or she has made at least one booking on every listing for which a request has been made, the traveler qualifies as a committed traveler. Don't include travelers who have never made a booking request.

Attribute	
travelerID	ID of a committed traveler
surname	surname of the traveler
numListings	total number of listings booked
Order by	travelerID ascending
Everyone?	Include only travelers committed travelers
Duplicates?	There can be no duplicates.

7. **Bargainers** On BnB, homeowners can set their own prices and travelers can bargain for a lower price. A good bargainer is someone who pays more than 25% less than the average per night cost for a listing. To compute this, you will need to find the average per night price per listing. A booking from January 1, 2016 to January 3, 2016 includes two nights.

Your query should return anyone who is a good bargainer on at least three different listings. Return the traveler ID, the average price per night that they paid, and the largest bargain percentage that he or she ever got (along with the listingID on which this bargain was obtained). For example, if the average per night cost is $100 and a traveler paid $25 per night then this is a 75% bargain.

Attribute	
travelerID	traveler ID
largestBargainPercentage	percentage less than the average nightly price percentage should be an integer between 0 and 100
listingID	listing on which the largest bargain percentage was received
Order by	largestBargainPercentage descending, then travelerID ascending
Everyone?	only travelers satisfying query
Duplicates?	a traveler is reported at most once

8. **Scratching backs?** We want to know how the ratings that a traveler gives (to a homeowner) compare to the ratings that traveler gets. Let's say there is a reciprocal rating for a booking if both the homeowner rated the traveler for that booking and the traveler rated the homeowner for that booking.

   For each traveler report the number of reciprocal ratings they have and the number of reciprocal ratings that differ by one point or less.

Attribute	
travelerID	ID of a traveler
reciprocals	number of reciprocal ratings they have
backScratches	number of reciprocal ratings that differ by one point or less
Order by	number of reciprocals, and in the case of ties, by number of backScratches (both descending)
Everyone?	include all travelers
Duplicates?	There can be no duplicates

Part 2: Embedded SQL

Imagine a BNB app used by travelers and homeowners. 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 or output. Doing so even once will result in the autotester terminating, causing you to receive a **zero** for this part.
- 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 may not change the method signature of any of the methods we've asked you to implement. However, you are welcome to write helper methods to maintain good code quality.
- As you saw in lecture, to run your code, you will need to include the JDBC driver in your class path. You may wish to review the related JDBC Exercise posted on the course website.

Your task

Open the starter code in `Assignment2.java`, and complete the following methods.

1. `connectDB`: Connect to a database with the supplied credentials.
2. `disconnectDB`: Disconnect from the database.
3. `homeownerRecommendation`: A method that, given a homeowner, returns the 10 most similar homeowners based on traveler reviews. See section Homeowner Recommendation below for details.
4. `booking`: A method that would be called when a homeowner accepts a booking request.

You will have to decide how much the database will do and how much you'll 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, and part of your mark for Part 2 will be based on whether or not you do so.

We 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.

Homeowner Recommendation

Method `homeownerRecommendation` must find homeowners who are similar to a given homeowner. This can be used by BnB to recommend particular homeowners to a traveler — homeowners that are similar to the ones that he or she rates highly. (Because we are only finding similar homeowners, for this part of the assignment we will only consider traveler's ratings of homeowners. We will not consider homeowners ratings of travelers.)

Here we specify how you are to determine similarity of homeowners.

In a recommendation system, there are two classes of entities, which we shall refer to as travelers and homeowners (alternatively, in an e-commerce application like Amazon, the entities might be customers and products). Travelers have ratings (preferences) for certain homeowners. For this part of the assignment, a traveler's rating of a homeowner is the average rating they have given the homeowner for all bookings they have made on a property (listing) owned by the homeowner.

The ratings are represented as a *utility matrix*, which for each traveler-homeowner pair, gives a value that represents what is known about the degree of preference (rating) of that traveler for that homeowner. The utility matrix is sparse, meaning that most entries are unknown. An unknown rating implies that we have no explicit information about the traveler's rating of a homeowner. In other words, if a traveler T has never made a booking in a listing owned by a homeowner H, or has never rated such a listing, then there is no rating of T for H.

Your homeowner recommender should compute the similarity between a given homeowner (the query homeowner) and others using *Cosine similarity*, which is the dot product between the columns representing different homeowners. A higher value indicates a greater degree of similarity. Your recommender should then return the top-10 most similar homeowners to the query homeowner (meaning the 10 homeowners with the highest similarity to the query homeowner).

Example: Consider the following ratings by travellers of homeowners. Many of the entries will be null ($\emptyset$) and these nulls will be treated as zero in the similarity computation.

	Homeowner			
Traveler	Jon Snow	Tyrion Lannister	Arya Stark	...
Daisy Mason	$\emptyset$	5	5	...
Violet Crawley	1.5	2	$\emptyset$	...
Tom Branson	4.5	$\emptyset$	5	...
...	...	...	...	...

The similarity of Jon Snow to Tyrion Lannister is $0 \times 5 + 1.5 \times 2 + 4.5 \times 0 = 3$. The similarity of Jon Snow to Arya Stark is $0 \times 5 + 1.5 \times 0 + 4.5 \times 5 = 22.5$. If we have a traveler who we know likes Jon Snow, then we could call your method `homeownerRecommendation`, which will compute the similarity between Jon Snow and all the other homeowners and return the top 10 homeowners who are most similar to Jon Snow.

In the event of a tie for the 10th position (only) you should report all ties; as a result there may be more than 10 homeowners returned. For example:

- If you have scores as follows you would only report H1... H10.
  Similarity scores: H1:20, H2:20, H3:20, H4:19, H5:19, H6:19, H7:19, H8:18, H9:18, H10:18, H11:17.
- If you have scores as follows you would report twelve homeowners H1... H12.
  Similarity scores: H1:20, H2:20, H3:20, H4:19, H5:19, H6:19, H7:19, H8:18, H9:18, H10:18, H11:18, H12:18, H13:17.

Additional 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 travelerId " +
"from traveler t join Booking b on t.travelerId = b.travelerId " +
"where city = 'Toronto'";
```

Here are some common mistakes and the error messages they generate:

- You forget the colon:

  ```
  wolf% java -cp /local/packages/jdbc-postgresql/postgresql-8.4-701.jdbc4.jar Example
  Error: Could not find or load main class Example
  ```

- You ran it on a machine other than dbsrv1

  ```
  wolf% java -cp /local/packages/jdbc-postgresql/postgresql-8.4-701.jdbc4.jar: Example
  SQL Exception.<Message>: Connection refused. Check that the hostname and port are correct and
  that the postmaster is accepting TCP/IP connections.
  ```