Copyright (c) 2020 Diane Horton

University of Toronto
CSC343, Fall 2020

Assignment 2

Due: Thursday 12 November, 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 the Toronto Public Library. Keep in mind that your code for this assignment must work on *any* database instance (including ones with empty tables) that satisfies the schema.

All values of type time, timestamp etc. in the dataset provided are on a 24-hour clock.

Begin by getting familiar with the schema that we have provided. Ask yourself questions like the ones laid out for you in Assignment 1. (By now, you don't need me to provide them.)

Part 1: SQL Queries

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 auto-tester (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 dependent 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.

Write SQL queries for each of the following:

1. **Branch Activity** We need a summary of activity across the branches to determine budget and staffing allocations. For every branch, and for each year from 2015 to 2019 inclusive, report the items below.

Attribute	
branch	The code for a branch.
year	A year between 2015 and 2019 inclusive.
events	The number of events at that branch during that year. If there were none, report 0 (not NULL).
sessions	The average number of sessions per event, across all events at that branch during that year. If there were none, report 0 (not NULL). If an event has a session that spans multiple years, the number is split up across the years.
registration	The total registration across all events at that branch during that year. If there were none, report 0 (not NULL). If an event has sessions spanning multiple years, the number of people registered for the event contributes to the total in each of those years.
holdings	The total number of different holdings at that branch. If there are none, report 0 (not NULL). Multiple copies of the same holding all count towards this total.
checkouts	The total number of checkouts at that branch during that year. If there are none, report 0 (not NULL).
duration	The average number of days between checkout and return for all items that were checked out during that year. Note: the return date could be in a subsequent year. Items that have not been returned do not contribute to duration. Report "N/A" if there not items that were checked out and returned.
Everyone?	For every branch, include every year from 2015 to 2019 inclusive.
Duplicates?	Not possible.

2. **Overdue items** These are the rules for due dates:

   - books and audiobooks: 21 days after checkout
   - movies, music, magazines and newspapers: 7 days after checkout

   Precision is on the scale of days. An item that was due yesterday or earlier and has not been returned yet is considered "overdue". An item that is due today is merely due, not overdue.

   For each branch in the ward "Parkdale-High Park", find all patrons with overdue items from that branch.

   To determine whether something is overdue, you will need to compare to today's date. Read the postgreSQL documentation to find out how to get today's date. Note that you can add an integer to a date to get a new date that is that many days ahead.

Attribute	
branch	The code for a branch.
email	The email address of a patron with an overdue item at this branch.
title	The title of the overdue item
overdue	The number of days the item is overdue.
Everyone?	Include every branch in Parkdale-High Park. For each, include only patrons who have an overdue item. If a branch has no overdue items, it will not appear.
Duplicates?	There can be no duplicate rows, but a branch can occur more than once as can a patron.

3. **Promotion** Let's define a patron's "libraries used" as the set of all libraries for which they have signed out at least one book and/or registered for at least one event.

Copyright (c) 2020 Diane Horton

The TPL wants to promote events to patrons based on their past library use, and has defined four categories for certain patrons of interest. These categories are based on two things: (1) how many books the patron has checked out:

- "low" means: their total number of books checked out is less than 25% of the average number of books checked out (from any library) across all patrons who have ever signed out a book at that patron's libraries used.
- "high" means: their total number of books checked out is more than 75% of the average number of books checked out (from any library) across all patrons who have ever signed out a book at that patron's libraries used.

and (2) how many events they have attended:

- "low" means: their total number of events attended is less than 25% of the average number of events attended (at any library) across all patrons who have ever attended an event at that patron's libraries used.
- "high" means: their total number of events attended is more than 75% of the average number of events attended (at any library) across all patrons who have ever attended an event at that patron's libraries used.

Combining these, we get four categories:

Attended:	Checked out:	
	low	high
low	inactive	reader
high	doer	keener

NB: Some patrons don't fall into any of the four categories.

For each patron, report their category as well as the total number of books they have taken out this year and the total number of events they have registered for this year.

Attribute	
patronID	the ID of this patron.
category	one of: "inactive", "reader", "doer", "keener". For patrons who fall in none of these categories, the value should be `null`.
Everyone?	Yes, include every patron.
Duplicates?	No, each patron should appear once in the results.

4. **Explorers contest**

   The library is considering a contest that will encourage people to get out and see more of the city. To enter the contest, a patron will be required to attend an event in some library in each ward, all within one calendar year. Some staff are sceptical that anyone would do this so they want to know if anyone has ever done it before.

   Find all patrons who have attended an event in some library in each ward, all within one calendar year. The library is calling them "explorers".

Attribute	
patronID	ID of a patron who meets the requirement in some year (or maybe several!)
Everyone?	Include only explorers.
Duplicates?	No.

5. **Lure them back.** The library wants to lure back patrons who formerly checked out many items, but whose checkouts have been diminishing.

   Define "patron active in month" to mean that that patron checked out at least one book from some branch that month.

   Find patrons who were active in every month of 2018, active in at least five months of 2019, had at least one month in 2019 when they were not active, and have checked out nothing in 2020.

Copyright (c) 2020 Diane Horton

Attribute	
patronID	ID of a patron who meets the criteria of this question.
email	Email address of this patron. If an email address is NULL, report it as “none”.
usage	Total number of different items the patron has ever checked out. An item that they have checked out multiple times only counts once.
decline	Difference between the number of checkouts out in 2018 and the number in 2019. Every checkout counts, even if the same item is checked out multiple times. A positive number indicates a decline. Note that the number could be 0 or even negative.
missed	The number of months in 2019 during which they checked out no items.
Everyone?	Include only patrons who meet the criteria of this question.
Duplicates?	No client can be included more than once.

6. **Devoted fans** Throughout this query we are considering only books, and only those that have a single contributor (who we'll refer to as the author).

   Let's say that patron P is a **devoted fan** of author A if and only if (1) P has checked out all of A's single-author books, or has checked out all but one of them, (2) P has given a review to all of A's single-author books that they have checked out, and (3) the average rating in those reviews was at least 4.0.

   For every patron, report the number of authors for whom they are a devoted fan.

Attribute	
patronID	ID of a patron
devotedness	The number of authors for whom they are a devoted fan.
Everyone?	Yes, include every patron.
Duplicates?	There can be no duplicates.

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 dealing with a value of type TIMESTAMP, DATE, or TIME, the EXTRACT function is handy for pulling out pieces.
- The CASE statement turns out to be quite useful.
- You may use any features defined in our version of postgreSQL on the Teaching Labs. This is not a pointed hint; I have not deliberately left features for you to discover that will dramatically simplify your work. However, I 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.
- 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 statements for each of the following:

Copyright (c) 2020 Diane Horton

1. **Out-of-bounds events** Find every event that has at least one session that is scheduled outside the hours of the library branch at which the event is held. Delete all such sessions, and if an event has no sessions left, then also delete the event and delete any registrations for it.

   None of this depends on whether the event was in the past. This means that you may erase information about something that actually happened. Don't try to avoid this.

2. **Auto-renew** The library is trying an experiment with auto-renewal for overdue books. For all books that are currently overdue at the Downsview branch, if the patron has no more than 5 books checked out and none is overdue by more than 7 days, extend the due date on all of the patron's books that are overdue at the Downsview branch by 14 days. Do it by changing the sign-out date to 14 days later.

3. **Extended hours** The library wants to make sure that every branch is open at least once a week outside of "business hours". For any branch that is closed on Sundays and is never open past 6pm on a weeknight, extend its hours on Thursdays to be open until 9 pm. Do not add a new row to its LibraryHours table; just modify the end time in the relevant row. But if a library branch is not already open on Thursdays, do add a row, and set the branch's Thursday hours to be 6 to 9pm.

Part 3: 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 methods that we have documented in the starter code in `Assignment2.java`.

1. `connectDB`: Connect to a database with the supplied credentials.
2. `disconnectDB`: Disconnect from the database.
3. `search`: A method that would be called when a patron searches for all holdings by a given contributor at a given branch.
4. `register`: A method that would be called when a patron registers for an event.
5. `item_return`: A method that would be called when a patron returns an item.

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 introduce any Java data structure such as an ArrayList, HashMap, or Set, or even a simple array. You will notice that we used an ArrayList in method `search`, but this is not so that it can perform computation on the ArralyList; it is only there so that the method could communicate its results to the user interface.

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.

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 distinct dept, cNum " +
   "from Offering o join Took t on o.oid = t.oid " +
   "where grade >= 95;" +
```

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