1. Preliminaries

Under Resources→Lab4 on Piazza, there are some files that are discussed in this document. Two of the files are lab4_ create.sql script and lab4_data_loading.sql. The lab4_ create.sql script creates all tables within the schema Lab4. The schema is (almost) the same as the one we used for Lab3, but there is no modifyVisit table. We included the Referential Integrity constraints from Lab1, but did not include the Referential Integrity constraints or General constraints from Lab3.

lab4_data_loading.sql will load data into those tables, just as a similar file did for Lab3. Alter your search path so that you can work with the tables without qualifying them with the schema name:

```
ALTER ROLE <username> SET SEARCH_PATH TO Lab4;
```

You must log out and log back in for this to take effect. To verify your search path, use:

```
SHOW SEARCH_PATH;
```

Note: It is important that you do not change the names of the tables. Otherwise, your application may not pass our tests, and you will not get any points for this assignment.

2. Instructions to compile and run JDBC code

Two important files under Resources→Lab4 are *RunRestaurantApplication.java* and *RestaurantApplication.java.* You should also download the file *postgresql-42.2.8.jar*, which contains a JDBC driver, from https://jdbc.postgresql.org/download/postgresql-42.2.8.jar

Place those 3 files into your working directory. That directory should be on your Unix PATH, so that you can execute files in it. Also, follow the instructions for "Setting up JDBC Driver, including CLASSPATH" that are at
https://jdbc.postgresql.org/documentation/head/classpath.html
Those instructions are hard to understand, so step-by-step instructions for setting up CLASSPATH appear in the last section (Section 8) of this document.

Modify *RunRestaurantApplication.java* with your own database credentials. Compile the Java code, and ensure it runs correctly. It will not do anything useful with the database yet, except for logging in and disconnecting, but it should execute without errors.

If you have changed your password for your database account with the "ALTER ROLE username WITH PASSWORD <new_password>;" command in the past, and you are using a confidential password (e.g. the same password as your Blue or Gold UCSC password, or your personal e-mail password), then be sure not to include this password in the *RunRestaurantApplication.java* file that you submit to us, as that information will be unencrypted.

You can compile the *RunRestaurantApplication.java* program with the following command (where the ">" character represents the Unix prompt):
> *javac RunRestaurantApplication.java*

To run the compiled file, issue the following command:
> *java RunRestaurantApplication*

Note that if you do not modify the username and password to match those of your PostgreSQL account in your program, the execution will return an authentication error. (We will run your program as ourselves, not as you, so we don't need to include your password in your solution.)

If the program uses methods from the *RestaurantApplication* class and both programs are located in the same folder, any changes that you make to *RestaurantApplication.java* can also be compiled with a javac command similar to the one above.

You may get ClassNotFound exceptions if you attempt to run your programs locally and there is no JDBC driver on the classpath, or unsuitable driver errors if you already have a different version of JDBC locally that is incompatible with *cse180-db.lt.ucsc.edu,* which is the class DB server. To avoid such complications, we advise that you use the provided *postgresql-42.2.5.jar* file, which contains a compatible JDBC library.

Note that Resources→Lab4 also contain a *RunFilmsApplication.java* file from an old 180 assignment; it won't be used in this assignment, but it may help you understand it, as we explain in Section 6.

3. Goal

The fourth lab project puts the database you have created to practical use. You will implement part of an application front-end to the database. As good programming practice, all of your methods should catch erroneous parameters, such as a value for *numMenuItemsOrdered* that's not positive in *getFrequentlyOrderedMenuItems*, and print out appropriate error messages.

4. Description of methods in the RestaurantApplication class

RestaurantApplication.java contains a skeleton for the *RestaurantApplication* class, which has methods that interact with the database using JDBC.

The methods in the *RestaurantApplication* class are the following:

- *getFrequentlyOrderedMenuItems:* This method has an integer argument called *numMenuItemsOrdered,* and returns the menuItemID for each menuItem where the total quantity (adding up quantity across all billEntry tuples) of that menuItem is greater than or equal to *numMenuItemsOrdered*. A value of *numMenuItemsOrdered* that's not positive is an error.

- *updateServerName:* Sometimes a server wants to change their name. The *updateServerName* method has two arguments, an integer argument, theServerID, and a string argument, newServerName. For the tuple in the server table (if any) whose serverID equals theServerID, *updateServerName* should update name to be newServerName. (Note that there may not be any tuples whose serverID matches theServerID.) *updateServerName* should return the number of tuples that were updated, which will always be 0 or 1.

- *reduceSomeVisitCosts*: This method has an integer parameters, maxVisitCount. It invokes a stored function *reduceSomeCostsFunction* that you will need to implement and store in the database according to the description in Section 5. *reduceSomeCostsFunction* should have the same parameters, *maxVisitCount*. A value of *maxVisitCount* that's not positive is an error.

  The visits table has a cost attribute. *reduceSomeCostsFunction* will reduce the cost for some (but not necessarily all) visits; Section 5 explains which visits should have their cost reduced, and also tells you how much they should be reduced. The *reduceSomeVisitCosts* method should return the same integer result that the *reduceSomeCostsFunction* stored function returns.

  The *reduceSomeVisitCosts* method must only invoke the stored function *reduceSomeCostsFunction*, which does all of the assignment work; do not implement the *reduceSomeVisitCosts* method using a bunch of SQL statements through JDBC.

Each method is annotated with comments in the RestaurantApplication.java file with a description indicating what it is supposed to do (repeating most of the descriptions above). Your task is to implement methods that match the descriptions. The default constructor is already implemented.

For JDBC use with PostgreSQL, the following links should be helpful. Note in particular, that you'll get an error unless the location of the JDBC driver is in your CLASSPATH.

Brief guide to using JDBC with PostgreSQL:
https://jdbc.postgresql.org/documentation/head/intro.html
Setting up JDBC Driver, including CLASSPATH:
https://jdbc.postgresql.org/documentation/head/classpath.html
Information about queries and updates:
https://jdbc.postgresql.org/documentation/head/query.html
Guide for defining stored procedures/functions:
https://www.postgresql.org/docs/11/plpgsql.html

5. Stored Function

As Section 4 mentioned, you should write a stored function called *reduceSomeCostsFunction* that has an integer parameters, maxVisitCount. *reduceSomeCostsFunction* will change the cost attribute for some (but not necessarily all) visit tuples. But it never reduces the costs of more than maxVisitCount visit tuples.

The customer table has attributes including customerID and status. We say that a customer has High, Medium, Low status or NULL status based on whether that customer's status is 'H', 'M', 'L" or NULL. The visit table has attributes including customerID and cost. We're going to reduce the cost for visits of customers based on their status **(but only if that cost is NOT NULL)**.

- We'll reduce the cost of visits by High status customers by 10%
- We'll reduce the cost of visits by Medium status customers by 5%.
- We'll reduce the cost of visits by Low status customers by 1%.
- We do nothing for the cost of visits for customers who have NULL status.

But we won't reduce the cost in all visit tuples; we'll only reduce the costs for at most maxVisitCount visit tuples. How do we decide which costs to reduce?

First, we process the visits by High status customers, ordered by increasing joinDate. Then we process the visits by Medium status customers, ordered by increasing joinDate. Then we process the visits by Low status customers, order by increasing joinDate. ("Processing" involves doing the reductions that are described above.) But as soon as we have processed maxVisitCount visit tuples, we are done. The value that *reduceSomeCostsFunction* returns is the number of visit tuples that have been updated.

Won't that value always equal maxVisitCount? No. Suppose that there are 3 visits by High status customers, 5 visits by Medium status customers, and 9 visits by Low status customers.

- If maxVisitCount is 17 or more, then all 17 of these visits will have their costs reduced, and *reduceSomeCostsFunction* returns the value 17 (even if maxVisitCount was 20).
- If maxVisitCount is 8, then the costs for the visits by the High and Medium status customers will be reduced, and *reduceSomeCostsFunction* returns the value 8.
- If maxVisitCount is 7, then the costs for the 3 visits by the High status customers will be reduced, and the costs for 4 of the 5 Medium status customer visits will be reduced, and *reduceSomeCostsFunction* returns the value 7. Which Medium status customers receive the reduction? The ones with the earlies joinDate. (Don't worry about multiple tuples that have the same joinDate; if there are 3 tuples that have the same joinDate and you're only allowed to reduce cost for 2 of them, any 2 of them are okay.)

Write the code to create the stored function, and save it to a text file named *reduceSomeCostsFunction.pgsql.* To create the stored function *reduceSomeCostsFunction*, issue the psql command:

```
\i reduceSomeCostsFunction.pgsql
```

at the server prompt. If the creation goes through successfully, then the server should respond with the message "CREATE FUNCTION". You will need to call the stored function within the *reduceSomeVisitCosts* method through JDBC, as described in the previous section, so you'll need to create the stored function before you run your program. You should include the *reduceSomeCostsFunction.pgsql* source file in the zip file of your submission, together with your versions of the Java source files *RestaurantApplication.java* and *RunRestaurantApplication.java* that were described in Section 4.

As we noted above, a guide for defining stored functions with PostgreSQL can be found here on the PostgreSQL site. PostgreSQL stored functions have some syntactic differences from the PSM stored procedures/functions that were described in class, and PostgreSQL. For Lab4, you should write a stored function that has only IN parameters; that's legal in both PSM and PostgreSQL.

We'll give you some more hints on Piazza about writing PostgreSQL stored functions.

6. Testing

The file *RunFilmsApplication.java* (this is not a typo) contains sample code on how to set up the database connection and call application methods **for a different database and for different methods**. *RunFilmsApplication.java* is provided only for illustrative purposes, to give you an idea of how to invoke the methods that you want to test in this quarter's assignment. It is not part of your Lab4 assignment, so it should not be submitted as part of your solution. Moreover, *RunFilmsApplication.java* won't compile successfully, since we haven't provided the *FilmsApplication.java* file that it uses.

RunRestaurantApplication.java is the program that you will need to modify in ways that are similar to the content of the *RunFilmsApplication.java* program, but for this assignment, not for a Films-related assignment. You should write tests to ensure that your methods work as expected. In particular, you should:

- Write one test of the *getFrequentlyOrderedMenuItems* method with the *numMenuItemsOrdered* argument set to 65. Your code should print the result returned as output. Remember that your method should run correctly for any value of *numMenuItemsOrdered*, not just when it's 65.

  You should also print a line describing your output in the Java code of *RunRestaurant*. The overall format should be as follows:

  ```
  /*
  * Output of getFrequentlyOrderedMenuItems
  * when the parameter numMenuItemsOrdered is 65.
  output here
  */
  ```

- Write two tests for the *updateServerName* method. The first test should be for theServerID 3 and newServerName 'Phileas Fogg'. The second test should be for theServerID 10 and newServerName 'John Smith'. Print out the result of *updateServerName* (that is the number of server tuples whose name attribute was updated) in *updateServerName* as follows:

  ```
  /*
  * Output of updateServerName when theServerID is 3
  * and newServerName is 'Phileas Fogg'
  output here
  */
  ```

  Also provide similar output for 10 and 'John Smith'.

- Also write two tests for the *reduceSomeVisitCosts* method. The first test should have maxVisitCount value 10. The second test should have maxVisitCount value 95. In *RunRestaurantApplication*, your code should print the result (total number of customers whose cost was updated) returned by each of the two tests, with a description of what each test was, just as for each of the previous methods.

  Please be sure to run the tests in the specified order, running with maxVisitCount 10, and then with maxVisitCount 95. The order affects your results.

Important: You must run all of these method tests in order, starting with the database provided by our create and load scripts. Some of these methods change the database, so using the database we've provided and executing methods in order is required.

7. Submitting

1. Remember to add comments to your Java code so that the intent is clear.
2. Place the java programs RestaurantApplication.java and RunRestaurantApplication.java, and the stored procedure declaration code *reduceSomeCostsFunction.pgsql* in your working directory at unix.ucsc.edu. Please remember to remove your password from RunRestaurantApplication.java before submitting.
3. Zip the files to a single file with name Lab4_XXXXXXX.zip where XXXXXXX is your 7-digit student ID, for example, if a student's ID is 1234567, then the file that this student submits for Lab4 should be named Lab4_1234567.zip. To create the zip file, you can use the Unix command:

   zip Lab4_1234567 RestaurantApplication.java RunRestaurantApplication.java reduceSomeCostsFunction.pgsql
4. Some students may want to use views to do Lab4. That's not required. But if you do use views, you must put the statements creating those views in a file called createRestaurantViews.sql, and include that file in your Lab4 zip file.
5. Lab4 is due on Canvas by 11:59pm on Tuesday, December 3, 2019. Late submissions will not be accepted, and there will be no make-up Lab assignments.

8. Step-by-step instructions for setting up CLASSPATH

These instructions are for students using the bash shell. If you are using csh instead, then use setenv instead of export to set CLASSPATH.

You will have to set the CLASSPATH to where your Lab4 files are stored. One simple approach is to download the JDBC jar file into your Lab4 directory, which we'll assume is ~/CSE180/lab4. (Each one's directory is different, e.g. /afs/cats.ucsc.edu/users/l/userid/CSE180/lab4/, which is equivalent to ~/CSE180/lab4/)

1. Check the current directory path using

```
pwd
```

   We're assuming that your directory is /afs/cats.ucsc.edu/users/l/userid/CSE180/lab4/, but it doesn't have to be. But you must use your directory consistently.

2. Then add the name of the jar file to the end of it, e.g.,

   /afs/cats.ucsc.edu/users/l/userid/CSE180/lab4/postgresql-42.2.8.jar

3. Append CLASSPATH to *.bash_profile*. It is located at ~/.bash_profile

```
vim ~/.bash_profile
```

   Then append the following command to at the end of *.bash_profile*. If you don't have a *.bash_profile*, 'vim ~/.bash_profile' will create a one for you.

```
export CLASSPATH=/afs/cats.ucsc.edu/users/l/userid/CSE180/lab4/postgresql-42.2.8.jar:.
```

   save the .bash_profile and execute commands from .bash_profile

```
source ~/.bash_profile
```

Be sure to use your own directory path. You could use a different directory, if you want, if you use it consistently.

Remember:

1. If you only run 'export CLASSPATH=/afs/cats.ucsc.edu/users/l/userid/CSE180/lab4/postgresql-42.2.8.jar:.', instead of writing it into *.bash_profile,* your CLASSPATH will be reset every time you close your terminal emulator.
2. You can check if you have set your CLASSPATH correctly using

```
echo $CLASSPATH
```

3. For csh, use setenv instead of export