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; there is no NewCustomers table, but there is a view BuyerSellerTotalCost (described below), and we also included referential integrity constraints. 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 *RunStockMarketApplication.java* and *StockMarketApplication.java.* You should also download the file *postgresql-42.2.5.jar*, which contains a JDBC driver, from https://jdbc.postgresql.org/download/postgresql-42.2.5.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

Modify *RunStockMarketApplication.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), make sure that you do not include this password in the *RunStockMarketApplication.jav*a file that you submit to us, as that information will be unencrypted.

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

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

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 *StockMarketApplication* class and both programs are located in the same folder, any changes that you make to *StockMarketApplication.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 *cmps180-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 *RunStoresApplication.java* file from an old CMPS 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 *numDifferentStocksSold* that's not positive in *getCustomersWhoSoldManyStocks*, and a value for *theCount* in *rewardBestBuyers* that's not positive, and print out appropriate error messages.

4. Description of methods in the StockMarketApplication class

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

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

- *getCustomersWhoSoldManyStocks:* This method has an integer argument called *numDifferentStocksSold,* and returns the customerID for each customer in Customers that has been the seller of at least *numDifferentStocksSold* different stocks in Trades. A value for *numDifferentStocksSold* that's not greater than 0 is an error.

- *updateQuotesForBrexit:* Quotes give information about stock prices. Brexit, the British exit from the European Union, may make a stock exchange that had expressed Quotes in euros to switch to British pounds instead. A euro is worth about 0.87, so exchanges will have to multiple prices in quotes by 0.87 to handle the conversion.

  The *updateQuotesForBrexit* method has one string argument, theExchangeID, which is the exchangeID for an exchange. (You don't have to test to see if there's an exchange in Exchanges which has that exchangeID.) *updateQuotesForBrexit* should update price in Quotes for every quote that has that exchangedID, multiplying price by 0.87. *updateQuotesForBrexit* should return the number of quotes whose prices were updated.

- *rewardBestBuyers*: This method has two integer parameters, theSellerID and theCount. It invokes a stored function *rewardBuyersFunction* that you will need to implement and store in the database according to the description in Section 5. *rewardBuyersFunction* should have the same two parameters, theSellerID and theCount. Trades has a volume attribute. *rewardBuyersFunction* will increase the volume for some trades whose sellerID is theSellerID; Section 5 explains which trade volumes should be increased, and how much they should be increased. The *rewardBestBuyers* method should return the same integer result as the *rewardBuyersFunction* stored function.

  The *rewardBestBuyers* method must only invoke the stored function *rewardBuyersFunction*, which does all of the assignment work; do not implement the *rewardBestBuyers* method using a bunch of SQL statements through JDBC. However, *rewardBestBuyers* should check to see whether theCount is greater than 0, and report an error if it's not.

Each method is annotated with comments in the StockMarketApplication.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/10/plpgsql.html

5. Stored Function

As Section 4 mentioned, you should write a stored function called *rewardBuyersFunction* that has two integer parameters, theSellerID and theCount. *rewardBuyersFunction* will increase the volume of some of the trades whose sellerID is theSellerID. But you'll only increase volume for at most theCount different buyers. (Let's hope that these buyers feel rewarded by this.)

The cost of a trade in Trades is price * volume. In lab4_create.sql, we provided a view, BuyerSellerTotalCost(buyerID, sellerID, totalCost), that gives the total cost of all the trades that took place for each buyer and seller. (Of course, if there were no trades between a buyer and seller, there is no tuple for that buyer/seller in BuyerSellerTotalCosts.) You will probably find it helpful to use that view in writing *rewardBuyersFunction*.

rewardBuyersFunction should **increase the volume in Trades** for some trades in which the seller was theSellerID. category is an attribute of Customers.

- If the buyer is a high status customer (category 'H'), then increase the volume by 50.
- If the buyer is a medium status customer (category 'M'), then increase the volume by 20.
- If the buyer is a low status customer (category 'L'), then increase the volume by 5.
- If the buyer has any other category, then increase the volume by 1.

However, you won't increase the volume of every trade in Trades for which sellerID is theSellerID; you'll adjust trades that involve at most theCount different buyers.

Which trades in Trades should have volume increases? Answer: The volume should be increased only for those buyers whose trades with theSeller had the highest totalCost (as shown in the view BuyerSellerTotalCost). For example, if theCount is 3:

a) If there are tuples in BuyerSellerTotalCost for theSellerID with 3 or more different buyers, then the volumes of all trades involving those 3 buyers with the highest totalCost should be increased (with the volume increase based on the category of the buyer).
b) If there are tuples in BuyerSellerTotalCost for theSellerID with only 2 different buyers, then the volumes of all trades involving those 2 buyers should be increased (with the volume increase based on the category of the buyer).
c) If there are no tuples in BuyerSellerTotalCost for theSellerID with any buyers, then there should be no volume increases.

The *rewardBuyersFunction* stored function should return the total number of trades for theSellerID whose volumes were increased. Note that there might be many trades whose volumes are increased even if theCount is 1.

If theCount is 3 and there are multiple buyers that have the third highest totalCost, then you may increase the volume of any one of them. But you should never increase volumes for more than theCount different buyers.

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

```
\i rewardBuyersFunction.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 *rewardBestBuyers* method through JDBC, as described in the previous section. You should include the *rewardBuyersFunction.pgsql* source file in the zip file of your submission, along with your versions of the Java source files *StockMarketApplication.java* and *RunStockMarketApplication.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 *RunStoresApplication.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**. *RunStoresApplication.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 assignment. It is not part of your Lab4 assignment, so it should not be submitted as part of your solution.

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

- Write one test of the *getCustomersWhoSoldManyStocks* method with the *numDifferentStocksSold* argument set to 3. Your code should print the result returned as output. Remember that your method should run correctly for any value of *numDifferentStocksSold*, not just when it's 3.

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

  ```
  /*
  * Output of getCustomersWhoSoldManyStocks
  * when the parameter numDifferentStocksSold is 3.
  output here
  */
  ```

- Write one test for the *updateQuotesForBrexit* method that updates prices quotes whose exchangeID is 'LSE '. (which is six characters, including 3 spaces at the end). Print out the result of *updateQuotesForBrexit* (that is the number of stocks whose price was updated) in *RunStockMarketApplication* as follows:

  ```
  /*
  *Output of updateQuotesForBrexit when theExchangeID is 'LSE '
  output here
  */
  ```

- Write two tests for the *rewardBestBuyers* method. The first test should have theCount value 2 and theSellerID value 1456. The second test should have theCount value 4 and theSellerID value 1456 (yes, the same value of theSellerID as in the first test). In *RunStockMarketApplication*, your code should print the result (total number of trades involving theSellerID whose volume was increased) returned by each of the two tests, with a description of what each test was, just as for the previous methods.

  Please be sure to run the tests in the specified order, running with theCount 2, and then with theCount 4. 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 might matter.

7. Submitting

1. Remember to add comments to your Java code so that the intent is clear.
2. Place the java programs StockMarketApplication.java and RunStockMarketApplication.java, and the stored procedure declaration code *rewardBuyersFunction.pgsql* in your working directory at unix.ucsc.edu. Please remember to remove your password from RunStockMarketApplication.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 StockMarketApplication.java RunStockMarketApplication.java rewardBuyersFunction.pgsql

4. Lab4 is due on Canvas by 11:59pm on Wednesday, March 13, 2019. Late submissions will not be accepted, and there will be no make-up Lab assignments.