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 similar to the one in our create.sql solution to Lab2, except that the columns and constraints Offers table has been changed, as described in a Piazza note posted on Saturday, November 28. We included all the constraints that were in our Lab2 solution (with changes to the Offers table). Lab3's new General constraints and revised Referential Integrity constraints are not in the schema.

lab4_data_loading.sql loads data into those tables, just as similar files did for previous Lab Assignments. 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 for database access from C

An important files under Resources→Lab4 is *runRealEstateApplication.c*. That file is not compilable as is. You will have to complete it to make it compilable, including writing three C functions that are described in Section 4 of this document. You will also have to write a Stored Function that is used by one of those C functions; that Stored Function is described in Section 5 of this document. As announced at the beginning of the quarter, we assume that CSE 180 students are familiar with C. However, you will not have to use a Make file, since *runRealEstateApplication.c* is the only file in your C program.

Assuming that *runRealEstateApplication.c* is in your current directory, you can compile it with the following command (where the ">" character represents the Unix prompt):

> *gcc -L/usr/include -lpq -o runRealEstateApplication runRealEstateApplication.c*

When you execute *runRealEstateApplication*, its parameters will be your userid and your password for our PostgreSQL database. So after you have successfully compiled your *runRealEstateApplication.c* (and separately successfully created your Stored Function in the database), then you'll be able to execute your program by saying:

> runRealEstateApplication <your_userid> <your_password>

[Do not put your userid or your password in your program code, and do not include the < and > symbols, just the userid and password. We will run your program as ourselves, not as you.]

How you develop your program is up to you, but we will run (and grade) your program using these commands on unix.ucsc.edu. So you must ensure that your program works in that environment. Don't try to change your grade by telling us that your program failed in our environment, but worked in your own environment; if you do, we'll point you to this Lab4 comment.

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, your functions should catch erroneous parameters, such as a value for theBrokerLevel that's not 'A', 'B', 'C' or 'D' in *getBrokerLevelCount*. For Lab4, if a parameter supplied to one of your functions is invalid, you should print an appropriate error message, and call exit(EXIT_FAILURE). In practice, application code should be more robust than that, continuing execution despite erroneous parameters. But for Lab4 we're keeping things simple and exiting, since your runRealEstateApplication code is supposed to be executing your functions with valid parameters.

4. Description of the three C functions in the runRealEstateApplication file that interact with the database

The *runRealEstateApplication.c* that you've been given contains skeletons for three C functions that interact with the database using libpq.

These three C functions are described below. The first argument for all of these C function is your connection to the database.

- *getBrokerLevelCount:* brokerLevel is an attribute in the Brokers table. Besides the database connection, the *getBrokerLevelCount* function has a char argument, theBrokerLevel, and it should return the number of Brokers whose brokerLevel value equals theBrokerLevel. A value of theBrokerLevel that is not 'A', 'B', 'C' or 'D' is an error, and you should exit.

- *updateCompanyName:* Brokers work at companies. The company that a broker works at is indicated by the companyName attribute in the Brokers table Sometimes the company that a broker works at changes, perhaps the original company was bought by another company.

  Besides the database connection, the *updateCompanyName* function has two arguments, a string argument oldCompanyName and another string argument, newCompanyName. For every broker in the Brokers table (if any) whose companyName equals oldCompanyName, *updateCompanyName* should update their companyName to be newCompanyName. (Of course, there might not be any tuples whose companyName matches oldCompanyName.) *updateCompanyName* should return the number of tuples that were updated (which might be 0).

- *increaseSomeOfferPrices*: Besides the database connection, this function has two integer parameters, theOffererID, and numOfferIncreases. *increaseSomeOfferPrice* invokes a Stored Function, *increaseSomeOfferPricesFunction*, that you will need to implement and store in the database according to the description in Section 5. The Stored Function *increaseSomeOfferPricesFunction* should have the same parameters, theOffererID and numOfferIncreases as *increaseSomeOfferPrices*. A value of numOfferIncreases that's not positive is an error, and you should exit.

  An offerer is a person who made an offer trying to buy a house that's for sale. The Offers table has an offererID attribute, which gives the ID of the offerer, and an offerPrice attribute, which gives the offer price that's the offerer made because they want to buy the house. *increaseSomeOfferPricesFunction* will increase the offerPrice for some (but not necessarily all) of the Offers made by theOffererID. Section 5 explains which Offers should have their offerPrice increased, and also tells you how much you should increase those offerPrice values. The *increaseSomeOfferPrices* function should return the same integer result that the *increaseSomeOfferPricesFunction* Stored Function returns.

  The *increaseSomeOfferPrices* function must only invoke the Stored Function *increaseSomeOfferPricesFunction*, which does all of the work for this part of the assignment; *increaseSomeOfferPrices* should not do the work itself.

Each of these three functions is annotated in the runRealEstateApplication.c file we've given you, with comments providing a description of what it is supposed to do (repeating the above descriptions). Your task is to implement functions that match those descriptions.

The following helpful libpq-related links appear in Lecture 11.

- Chapter 33. libpq - C Library in PostgreSQL docs, particularly:
  - 31.1. Database Connection Control Functions
  - 31.3. Command Execution Functions
  - 33.21. libpq Example Programs
- PostgreSQL C tutorial from Zetcode

5. Stored Function

As Section 4 mentioned, you should write a Stored Function (not a Stored Procedure) called *increaseSomeOfferPricesFunction* that has two integer parameters, theOffererID, and numOfferIncreases. *increaseSomeOfferPricesFunction* will increase offerPrice by 8000 for some tuples in the Offers table whose offererID attribute value equals theOffererID. It should not change offerPrice values for any other offerer. This Stored Function should count the number of the increases that it has made, and it should return this count. However, this count should not be more than numOfferIncreases. (It might equal numOfferIncreases; it might be less.)

Here's how to determine which Offers should have their offerPrice increase by 8000:

- Only increase offerPrice for Offers made when offererID equals theOffererID.
- Note that Offers can only be made on ForSaleHouses (referential integrity constraint). Some of the houses on which theOffererID has made Offers have been sold, meaning that there is a tuple in SoldHouses that matches the Primary Key of ForSaleHouses. Do not increase offerPrice for any "for sale house" that has been sold.
- offerPrice for any house that theOffererID wants to buy but which have not been sold might be increased by 8000. To determine which offerPrice values should be increased, order these houses by offerPrice in ascending order, so that the smallest offerPrice value comes first. The first numOfferIncreases houses according to that ordering should have offerPrice increased by 8000.
    - What happens if there are more than numOfferIncreases houses? Only the first numOfferIncreases offerPrice values are increased, and the value returned is numOfferIncreases.
    - What happens if there are exactly numOfferIncreases houses? Those numOfferIncreases offerPrice values are increased, and the value returned is numOfferIncreases.
    - What happens if there are fewer than numOfferIncreases houses? offerPrice values for those houses are increased, and the value returned is the actual number of houses whose offerPrice was increased, which will be less than numOfferIncreases.

Write the code to create the Stored Function, and save it to a text file named *increaseSomeOfferPricesFunction.pgsql.* To create the Stored Function *increaseSomeOfferPricesFunction*, issue the psql command:

```
\i increaseSomeOfferPricesFunction.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 from the *increaseSomeOfferPrices* function in your C program, as described in the previous section, so you'll need to create the Stored Function before you run your program. You should include the increaseSomeOfferPricesFunction.pgsql source file in the zip file of your submission, together with your versions of the C source files runRealEstateApplication.c that was described in Section 4; see Section 7 for detailed instructions.

A guide for defining Stored Functions for 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 Lecture. 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

Within main for *runRealEstateApplication.c*, you should write several tests of the C functions described in Section 4. You might also want to write your own tests, but the following tests must be included in the *runRealEstateApplication.c* file that you submit in your Lab4 solution.

- Write one test of the *getBrokerLevelCount* function with the theBrokerLevel argument set to 'B'. Your code should print the result returned as output. Remember that your function should run correctly for any legal value of theBrokerLevel, not just for broker level 'B'.

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

  ```
  /*
  * Output of getBrokerLevelCount
  * when the parameter theBrokerLevel is 'B'.
    < place your output here>
  */
  ```

- Write two tests for the *updateCompanyName* function. The first test should be for oldCompanyName 'Weathervane Group Realty' and newCompanyName 'Catbird Estates'. The second test should be for oldCompanyName 'Intero' and newCompanyName 'Sotheby'. Print out the result of *updateCompanyName* (that is the number of Brokers tuples whose companyName attribute was updated) as follows:

  ```
  /*
  * Output of updateCompanyName when oldCompanyName is
  * 'Weathervane Group Realty' and newCompanyName is 'Catbird Estates'
    < place your output here>
  */
  ```

  Also provide similar output for 'Intero' and 'Sotheby'.

- Also write two tests for the *increaseSomeOfferPrices* function. The first test should have theOffererID value 13 and numOfferIncreases value 4. The second test should have theOffererID value 13 and numOfferIncreases value 2, and it is run on the data resulting from the first test. In *runRealEstateApplication*, your code should print the result (number of 8000 increases that you made) returned by each of the two tests, with a description of what each test was, just as for each of the previous function. (The output format is up to you, as long as it provides the required information.)

  Please be sure to run the tests in the specified order, running first with numOfferIncreases 4, and then with numOfferIncreases 2. (Both tests are run for theOffererID 13.)

Important: You must run all of these function tests in order, starting with the database provided by our create and load scripts. Some of these function change the database, so using the database we've provided and executing functions in order is required. Reload the original load data before you start, but you do not have to reload the data multiple times.

7. Submitting

1. Remember to add comments to your C code so that the intent is clear.
2. Place the C program runRealEstateApplication.c and the stored procedure declaration code *increaseSomeOfferPricesFunction.pgsql* in your working directory at unix.ucsc.edu.
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 runRealEstateApplication.c increaseSomeOfferPricesFunction.pgsql

   Please do not include any other files in your zip file, except perhaps for an optional README file, if you want to include additional information about your Lab4 submission.
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 createRealEstateViews.sql, and include that file in your Lab4 zip file.
5. Lab4 is due on Canvas by 11:59pm on Sunday, December 13, 2020. Late submissions will not be accepted, and there will be no make-up Lab assignments.