1. Preliminaries

Under Resources→Lab4 on Piazza, there are some files that are discussed in this document. Two of the files are create_lab4.sql script and load_lab4.sql. The create._lab4.sql creates all tables within the schema Lab4. The schema is the same as our create_lab1.sql solution to Lab1; it has all the constraints that were in the Lab1 solution. Lab3's new General constraints and revised Referential Integrity constraints are not in this schema

load_lab4.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 (and view) 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 file under Resources→Lab4 is *runNewspaperApplication.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 *runNewspaperApplication.c* is the only file in your C program.

Assuming that *runNewspaperApplication.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 runNewspaperApplication runNewspaperApplication.c*

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

> runNewspaperApplication <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.

Because Lab4 is mainly a database application programming assignment, rather than a C language assignment, please don't use a makefile or a header (.h) file in Lab4.

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 (and impossible situations). We describe erroneous parameters and required error handling below.

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

The *runNewspaperApplication.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.

- *countCoincidentSubscriptions:* The Subscriptions table has an attribute subscriberPhone which identifies the subscriber who has that subscription. Besides the database connection, the *countCoincidentSubscriptions* function has one parameter, an integer, theSubscriberPhone.

  subscriptionStartDate is an attribute in the Subscriptions table. Even though subscriptionEndDate is not an attribute in the Subscriptions table, Lab2 Query4 told you how you can determine the end date for a subscription.

  subscriptionInterval is an attribute in SubscriptionKinds. The end date of a subscription can be computed by adding its subscriptionStartDate to the subscriptionInterval for its subscription kind.

  But when you add a DATE and an INTERVAL, the result is a TIMESTAMP (not a DATE). If you have a TIMESTAMP value myTimestamp, then one of the ways that you get the date from that TIMESTAMP is by writing DATE(myTimestamp).

  A particular subscriber may have multiple subscriptions. We'll say that two subscriptions (which we'll call s1 and s2) "coincide" if the start date for one of the two subscriptions (s2) occurs between the start date and end date of the other subscription (s1):

  1) s1's start date <= s2's start date, and
  2) s2's start date <= s1's end date.

  Note that in this example, there are 2 coincident subscriptions, since s1 coincides with s2, and s2 coincides with s1. (It doesn't matter which starts first.)

  The function *countCoincidentSubscriptions* should count how many subscriptions a particular subscriber (identified by parameter thesubscriberPhone) has that are coincident with at least one of that subscriber's other subscriptions. *countCoincidentSubscriptions* should return that count.

  In the example given above, *countCoincidentSubscriptions* should return the value 2, counting both s1 and s2. As another example, if a subscriber has 4 subscriptions each of which coincides with all of the others, the value returned should be 4 (not 6 or 12), since each one of the 4 subscriptions coincides with at least one other subscription.

  But be careful. It is possible that isn't a subscriber whose subscriberPhone value equals theSubscriberPhone. In that case, *countCoincidentSubscriptions* should -1. And it possible that there is a subscriber whose subscriberPhone value equals theSubscriberPhone, but that subscriber has no subscriptions. In that case, subscriber whose subscriberPhone value equals theSubscriberPhone should return 0 (since that

subscriber has no coincident subscriptions.)

Note that C functions can have more than one SQL statement in them. You'll have more than one SQL statement in your code for *countCoincidentSubscriptions*. And those multiple statements must be within a single Serializable transaction.

- *changeAddresses:* Sometimes, streets and cities change their names. The Subscribers table has an attribute subscriberAddress. Besides the database connection, *ChangeAddresses* has two other parameters, oldAddress and newAddress, in that order. There may be multiple subscribers who have the same address. *ChangeAddresses* should modify all the subscriberAddress values which equal oldAddress, changing them to newAddress. Moreover, ChangeAddresses should determine the number of subscribers whose subscriberAddress values were modified by ChangeAddresses, and return that value. .

  If there are no subscribers whose subscriberAddress equals oldAddress, that's <u>not</u> an error; in that case, *ChangeAddresses* should return 0, since no SubscriberAddress values were modified.

- *increaseSomeRates*: Besides the database connection, this function has one integer parameter, maxTotalRateIncrease. *increaseSomeRates* invokes a Stored Function, *increaseSomeRatesFunction*, that you will need to implement and store in the database according to the description in Section 5. The Stored Function *increaseSomeRatesFunction* should have the same maxTotalRateIncrease parameter that was supplied to *increaseSomeRates* (but the Stored Function does not have the database connection as a parameter).

  increaseSomeRatesFunction will increase rates for some (but not necessarily all) SubscriptionKinds in the SubscriptionKinds table. Section 5 explains which rates should be increased

  The *increaseSomeRates* function should return the same integer result that the *increaseSomeRatesFunction* Stored Function returns. *increaseSomeRatesFunction* may return a negative value, signifying an erroneous argument. *increaseSomeRates* should just return that value to its invoker, which will deal with that value as described in the Testing section below in Section 6.

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

Each of these three functions is annotated in the runNewspaperApplication.c file we've given you, with comments providing a description of what it is supposed to do (repeating just part of 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 *increaseSomeRatesFunction* which has one integer parameter, maxTotalRateIncrease. If maxTotalRateIncrease is not positive, then *increaseSomeRatesFunction* should return the value -1, signifying an error. (Of course, 0 is not positive.)

Our newspaper, the Daily Planet, is losing money, so we're going to increase the rate for some kinds of subscriptions (SubscriptionKinds). For each subscription kind, there may be 0 or more subscribers. We're going to look at the number of subscribers for each subscription kind (which we'll call the "popularity" of that subscription kind) to determine how much to increase the rate for that subscription kind.

- If a subscription kind has popularity 5 or more, the increase should be 10.
- If a subscription kind has popularity 3 or 4, the increase should be 5.
- If a subscription kind has popularity 2, the increase should be 3.
- If a subscription kind has popularity 0 or 1, there should be no increase.

So for example, if there are two subscription kinds which have popularity 6, one subscription kind which has popularity 5, three subscription kinds which have popularity 3, four subscription kinds which have popularity 2, and 7 subscription kinds which have popularity 1, then it's possible that the total rate increase our newspaper would make would be:

2*10 + 1*10 + 3*5 + 4*3 + 7*0 = 57

because the two subscription kinds with popularity 6 would have their rates increased by 10, the one subscription kind with popularity 5 would also have its rate increase by 10, the three subscription kinds with popularity 3 would have their rates increased by 5, and the four subscription kinds with popularity 2 would have their rates increased by 3.

However, *increaseSomeRatesFunction* has a parameter, maxTotalRateIncrease, and our newspaper's total rate increase is not allowed to be more than maxTotalRateIncrease. So *increaseSomeRatesFunction* should not always increase all of these rates. We'll keep track of the total rate increase we've computed so far for subscription kinds, starting with the most popular subscription kind. We'll impose a rate increase on a subscription kind only if the new total rate increase doesn't exceed maxTotalRateIncrease. The value that *increaseSomeRatesFunction* returns should be the total rate increase that we've made, which might be maxTotalRateIncrease, or might be less than maxTotalRateIncrease.

Let's describe what should happen for the example described above, when some different values of maxTotalRateIncrease supplied.

- If maxTotalRateIncrease is 60, then all of the rate increases would be made, so we return **57** = 2*10 + 1*10 + 3*5 + 4*3.
- If maxTotalRateIncrease is 55, then we can do the 10 rate increase for the two subscription kinds which have popularity 6 and also the subscription kind which has popularity 10 (total is 30), and the 5 rate increase for the three subscription kinds which have popularity 3 (total is 45). We can do the 3 rate increase for just three of the four subscription kinds which have popularity 2, which makes the total 54. Doing the rate increase for the fourth subscription kind which has popularity 2 would make the total 57, which is more than 55). So we return **54** = 3*10 + 3*5 + 3*3.
- If maxTotalRateIncrease is 43 then we can do the 10 rate increase for the two subscription kinds which have popularity 6 and also for the subscription kind which has popularity 5 (total is 30), but we can only do the 5 rate increase for the two of the three subscription kinds which have popularity 3. That adds 10 to the total, making the total 40; adding 5 for the third subscription kind which has popularity 3 would make the total 45, which is more than 43. But we can do the 3 rate increase for one of the four subscription kinds which have popularity 2, adding 3 to the total, which becomes 43, . So we return **43** = 3*10 + 2*5 + 1*3.
- If maxTotalRateIncrease is 19 then we can do the 10 rate increase for only one of the two subscription kinds which have popularity 6 (because 20 is more than 10). We can't do the 10 rate increase for the subscription kind which has popularity 5. (We needed to consider popularity 6 before considering popularity 5). Now we can do the 5 rate increase for just one of the three subscription kinds which have popularity 3, making the total 15, and we can do the 3 rate increase for just one of the four subscription kinds which have popularity 2, making the total 18 So we return **18** = 1*10 + 1*5 + 1*3.
- If maxTotalRateIncrease is 7, then we can't do the 10 rates increases for the subscription kinds which have popularity 6 or 5. We can do the 5 rate increase for just one of the subscription kinds that has popularity 3. But we can't do the 3 rate increase for any of the subscription kinds which have popularity 2 (since that would make the total 8). So we return **5** = 1*5.
- If maxTotalRateIncrease is 2, then none of the rate increases is possible, so we return **0**. (That's not an error.)

It should be clear that you need to iterate through the subscription kinds based on decreasing popularity. (You show be able to figure out how to do that.)

You might ask which subscription kind gets the rate increase if there are three subscription kinds that have the same popularity but you can only increase the rate for one of them. Doesn't matter; increase the rate for just one of them, but it can be any one of them.

Write the code to create the Stored Function, and save it to a text file named *increaseSomeRatesFunction.pgsql.* After you've saved that file, you can create the Stored Function *increaseSomeRatesFunction* by issuing the psql command:

```
\i increaseSomeRatesFunction.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 *increaseSomeRates* 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 increaseSomeRatesFunction.pgsql source file in the zip file of your submission, together with your versions of the C source files runNewspaperApplication.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, but there a better description on this PostgreSQL Tutorial site. For Lab4, you should write a Stored Function that has an IN parameter.

We've given you some more hints in Lecture 10 and on Piazza about writing PostgreSQL Stored Functions, including:

- *fireSomePlayersFunction.pgsql*, an example of a PostgreSQL Stored Function from another quarter, and
- *What_Does_fireSomePlayersFunction_Do.pdf*, an explanation of what that Stored Function does. But we won't provide the tables and load data for running *fireSomePlayersFunction.pgsql*.

6. Testing

Within main for *runNewspaperApplication.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 *runNewspaperApplication.c* file that you submit in your Lab4 solution.

All output lines should be on new lines. You may insert additional new lines if you like.

- Write 4 tests of the *countCoincidentSubscriptions* function.
  - The first test should be theSubscriber value 8315512.
  - The second test should be theSubscriber value 8313293.
  - The third test should be theSubscriber value 1234567.
  - The fourth test should be theSubscriber value 6502123.

  Printing occurs within main based on the value returned by *countCoincidentSubscriptions*, If the value it returns is 0, or more, then you should print out:

  Number of coincident subscriptions for <theSubscriberPhone> is <value returned>

  where <theSubscriberPhone> is the parameter to the function.

  But if *countCoincidentSubscriptions* returns -1, then print out:

  No subscriber exists whose phone is <theSubscriberPhone>

  where <theSubscriberPhone> is the parameter to the function. Continue executing additional tests if this occurs.

- Write 3 tests for the *ChangeAddresses* function.
  - The first test should be for oldAddress '100 Asgard St, Asgard, AG, 00001' and newAddress 'PQRS'.
  - The second test should be for oldAddress '3428A Lombard St, Tahoe City, CA, 96142' and newAddress 'ABCD'.
  - The third test should be for oldAddress 'IJL' and newAddress 'MNOP'.

  Your tests should print out:

  <value returned> addresses were modified by ChangeAddresses

  where <value returned> is the value returned by ChangeAddresses.

- Also write 5 tests for the *increaseSomeRates* function.
    - The first test should have maxTotalRateIncrease value 100.
    - The second test should have maxTotalRateIncrease value 45.
    - The third test should have maxTotalRateIncrease value 29.
    - The fourth test should have maxTotalRateIncrease value 2.
    - The fifth test should have maxTotalRateIncrease value 0.

  Run these tests and print their results from main in runNewspaperApplication. If *increaseSomeRates* returns a non-negative value, print out that result in the following format:

  Total increase for maxTotalRateIncrease <maxTotalRateIncrease> is <value returned>

  But increaseSomeRates can return a negative value, signifying an error. For these negative values, print an error message (format of error message is up to you) that describes the error and the erroneous parameter that resulted in the error, and exit (using bad_exit).

You must run all of these tests for all three of these functions in the specified order, starting with the database provided by our create and load scripts. Some of these functions modify the database, so be sure to use the load data that we've provided, executing the functions in the specified order. Reload the original load data before you start your tests, but you **do not** have to reload the data multiple times in Lab4.

Hmmm, do any of these tests affect each other? What do you think?

7. Submitting

1. Remember to add comments to your C code so that the intent is clear.
2. Place the C program runNewspaperApplication.c and the stored procedure declaration code *increaseSomeRatesFunction.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 runNewspaperApplication.c increaseSomeRatesFunction.pgsql

   Please do not include any other files in your zip file, except perhaps for a view creation file (described below) and an optional README file, if you want to include additional information about your Lab4 submission.
4. Some students might want to use additional 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 createNewspaperViews.sql which you include in your Lab4 zip file. Do not put your additional view inside create_lab4.sql, because Readers won't see them when grading.
5. Lab4 is due on Canvas by 11:59pm on Tuesday, December 5, 2023. Late submissions will not be accepted, and there will be no make-up Lab assignments.