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 the same as the one in our create.sql solution to Lab2. We included all the constraints that were in our Lab2 solution. Lab3's new General constraints and revised Referential Integrity constraints are not in the schema.

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 *RunZooApplication.java* and *ZooApplication.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 *RunZooApplication.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 *RunZooApplication.java* file that you submit to us, as that information will be unencrypted.

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

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

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 *ZooApplication* class and both programs are located in the same folder, any changes that you make to *ZooApplication.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 *cse182-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 contains a *RunFilmsApplication.java* file from an old 182 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, your methods should catch erroneous parameters, such as a value for *theMemberStatus* that's not 'A', 'B' or 'C' in *getMemberStatusCount*. For Lab4, if a parameter supplied to one of your methods is invalid, you should print an appropriate error message, and call System.exit(-1); 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 RunZooApplication code is supposed to be executing your methods with valid parameters.

4. Description of methods in the ZooApplication class

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

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

- *getMemberStatusCount:* This method has a string argument called *theMemberStatus,* and returns the number of Members whose memberStatus equals *theMemberStatus*. A value of *theMemberStatus* that's not 'L', 'M' or 'H' (corresponding to Low, Medium, and High) is an error.

- *updateMemberAddress:* Sometimes a member changes address. The updateMemberAddress method has two arguments, an integer argument theMemberID, and a string argument, newMemberAddress. For the tuple in the Members table (if any) whose memberID equals theMemberID, *updateMemberAddress* should update the address to be newMemberAddress. (Note that there might not be any tuples whose memberID matches theMemberID.) *updateMemberAddress* should return the number of tuples that were updated, which will always be 0 or 1.

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

  The Keepers table has a keeperSalary attribute, which gives the salary (in dollars and cents) for each keepers. *increaseSomeKeeperSalariesFunction* will increase the salary for some (but not necessarily all) keepers; Section 5 explains which keepers should have their salaries increased, and also tells you how much they should be increased. The *increaseSomeKeeperSalaries* method should return the same integer result that the *increaseSomeKeeperSalariesFunction* stored function returns.

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

Each method is annotated with comments in the ZooApplication.java file with a description indicating what it is supposed to do (repeating above descriptions). Your task is to implement methods that match those 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 *increaseSomeKeeperSalariesFunction* that has an integer parameter, maxIncreaseAmount. *increaseSomeKeeperSalariesFunction* will change the salary (keeperSalary attribute) for some (but not necessarily all) tuples in Keepers. This function should keep track of the total of the increases that it has made; that total should not be more than maxIncreaseAmount.

One of the attributes in the Keepers table is keeperLevel; another attribute is keeperSalary. Salaries will be increased only for keepers whose salary isn't NULL.

We say that a keeper is an 'A' keeper if their keeperLevel is 'A', and similarly for a 'B' keeper and a 'C' keeper. We're going to increase the keeperSalary for some keepers based on their keeperLevel.

- We'll increase the salary for 'A' keepers by $10.
- We'll increase the salary for 'B' keepers by $20.
- We'll increase the salary for 'C' keepers by $30.

But we won't increase the salary for all the keepers; the total of the increases can't be more than maxIncreaseAmount. How do we decide which salaries to increase?

First, we process the 'A' keepers, ordered by increasing hireDate. Then we process the 'B' keepers, ordered by increasing hireDate. Then we process the 'C' keepers, ordered by increasing hireDate. ("Processing" involves giving the increase described above, which may be $10, $20 or $30.) But we don't give an increase if the total of the increases would exceed maxIncreaseAmount. The value that *increaseSomeKeeperSalariesFunction* returns is the total of all the increases that *increaseSomeKeeperSalariesFunction* has made, which might be maxIncreaseAmount, or might be less than maxIncreaseAmount.

To see how this works, suppose that there are 4 'A' keepers, 6 'B' keepers, and 8 'C' keepers.

- If maxIncreaseAmount is 400 or more, then all 17 of these keepers will have their salaries increases, and *increaseSomeKeeperSalariesFunction* returns the value 400 (= 4*10 + 6*20 + 8*30), even if maxIncreaseAmount was 480.
- If maxIncreaseAmount is 240, then the salaries for all the 'A' and 'B' keepers will be increased, and so will the salaries of 2 of the 8 'C' keepers, and *increaseSomeKeeperSalariesFunction* returns the value 220 (= 4*10 + 6*20 + 2*30). Which 'C' keepers receive salary increases? The ones that have the earliest hireDate. (Don't worry about multiple 'C' keepers that have the same hireDate; if there are 3 'C' keepers that have the same hireDate, and you're only allowed to increase the salary for 2 of them, then any 2 of them are okay.)
- If maxIncreaseAmount is 100, then the salaries for all 4 'A' keepers will be increased, and the salaries for 3 of the 6 'B' keepers will also be increased, and *increaseSomeKeeperSalariesFunction* returns the value 100 = (4*10 + 3*20).

What if there are keepers whose keeperLevel has a value other than 'A', 'B' or 'C', such as NULL? The salaries of keepers who have levels other than 'A', 'B' or 'C' are not increased.

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

```
\i increaseSomeKeeperSalariesFunction.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 *increaseSomeKeeperSalaries* 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 *increaseSomeKeeperSalariesFunction.pgsql* source file in the zip file of your submission, together with your versions of the Java source files *ZooApplication.java* and *RunZooApplication.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.

RunZooApplication.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 *getMemberStatusCount* method with the *theMemberStatus* argument set to 'H'. Your code should print the result returned as output. Remember that your method should run correctly for any legal value of *theMemberStatus*, not just for member status 'H'.

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

  ```
  /*
  * Output of getMemberStatusCount
  * when the parameter theMemberStatus is 'H'.
    < place your output here>
  */
  ```

- Write two tests for the *updateMemberAddress* method. The first test should be for theMemberID 1006 and newMemberAddress '200 Rocky Road'. The second test should be for theMemberID 1011 and newMemberAddress '300 Rocky Road'. Print out the result of *updateMemberAddress* (that is the number of Members tuples whose name attribute was updated) in *updateMemberAddress* as follows:

  ```
  /*
  * Output of updateMemberAddress when theMemberID is 1006
  * and newMemberAddress is '200 Rocky Road'
    < place your output here>
  */
  ```

  Also provide similar output for 1011 and '300 Rocky Road'.

- Also write two tests for the *increaseSomeKeeperSalaries* method. The first test should have maxIncreaseAmount value 451. The second test should have maxIncreaseAmount value 132. In *RunZooApplication*, your code should print the result (total of all the salary 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 methods. (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 with maxIncreaseAmount 451, and then with maxIncreaseAmount 132.

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. 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 Java code so that the intent is clear.
2. Place the java programs ZooApplication.java and RunZooApplication.java, and the stored procedure declaration code *increaseSomeKeeperSalariesFunction.pgsql* in your working directory at unix.ucsc.edu. Please remember to remove your password from RunZooApplication.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 ZooApplication.java RunZooApplication.java increaseSomeKeeperSalariesFunction.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 createZooViews.sql, and include that file in your Lab4 zip file.
5. Lab4 is due on Canvas by 11:59pm on Tuesday, June 2, 2020 (yes, **Tuesday**). 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 indicate 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 ~/CSE182/lab4. (Each student's directory is different, e.g., /afs/cats.ucsc.edu/users/l/userid/CSE182/lab4/, which is equivalent to ~/CSE182/lab4/. You can use the pwd command to find out your directory.)

1. Check the current directory path using

```
pwd
```

   We're assuming that your directory is /afs/cats.ucsc.edu/users/l/userid/CSE182/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/CSE182/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/CSE182/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/CSE182/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