## 1. Preliminaries

Under Resources→Lab4 on Piazza, there are a number of files that are discussed in this document. Two of the files there are create_lab4.sql script and lab4_data_loading.sql. The create_lab4.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 NewTickets table. Also, to simplify things, the cost attribute in the Tickets table is now integer. 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

Three important files under Resources→Lab4 are *RunFlyingApplication.java*, *FlyingApplication.java* and *postgresql-42.1.4.jar*, which contains a JDBC driver. (There's also 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.) Place those three files into your working directory; that directory should be on your Unix PATH, so that you can execute files in it. Also, follow these instructions for "Setting up JDBC Driver, including CLASSPATH"

Modify *RunFlyingApplication.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 now use a confidentiality sensitive 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 *RunFlyingApplication.jav*a file that you submit to us, as that information will be unencrypted.

You can compile the *RunFlyingApplication.java* program with the following command:
*> javac RunFlyingApplication.java*

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

Note that if you do not modify the username and password to match those of your PostgreSQL account in the program, the execution will return an authentication error. If the program uses methods from the *FlyingApplication* class and both programs are located in the same folder, any changes that you make to *FlyingApplication.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-42.1.4.jar* file, which contains a compatible JDBC library.

## 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.

## 4. Description of methods in the FlyingApplication class

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

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

- *getAirlinesWithManyFlights:* This method has an integer argument called *numberOfFlights,* and returns the AirlineName (not the AirlineID) for each Airline that has at least *numberOfFlights* different Flights.

- *raiseAirlineTicketCosts:* This method takes an AirlineID and an increment as arguments, and raises the cost of all tickets that have that AirlineID by the amount specified in the increment argument. Of course, there may be many tickets that have that AirlineID. If there are no tickets with that name, r*aiseAirlineTicketCosts* should do nothing. The update should be performed as a single SQL statement. r*aiseAirlineTicketCosts s*hould return the number of tickets whose cost was incremented.

- *deleteSomeUnpaidTickets*: This method takes an integer deleteCount as input and invokes a stored function *deleteUnpaid* that you will need to implement and store in the database according to the description in Section 5. *deleteUnpaid* should delete some unpaid tickets; Section 5 explains which tickets the stored function should delete..

  *deleteSomeUnpaidTickets* must only invoke the stored function *deleteUnpaid*, which does all of the deletion work; do not implement this using a bunch of SQL statements through JDBC

Each method is annotated with comments in the FlyingApplication.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.

A brief guide to using JDBC with PostgreSQL can be found here on the PostgreSQL site, and information about queries and updates starts here.. This material should be useful when implementing the methods.

## 5. Stored Function

As Section 5 explains, you should write a stored function called *deleteUnpaid*, whose input is a deleteCount. For every customer whose Status is not 'A', this stored function should scan through the unpaid tickets for that customer and delete some of those tickets. Which of those tickets should be deleted?

- If deleteCount is 2, then the 2 most expensive unpaid tickets for that customer should be deleted, and similarly for other value of deleteCount.
- What if deleteCount is 2 and there is only 1 unpaid ticket for that customer? That 1 unpaid ticket should be deleted.
- Of course, if a customer has no unpaid tickets, then no tickets should be deleted.

The *deleteUnpaid* stored function should return the total cost (across all the customers) of the unpaid tickets that were deleted.

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

```
\i deleteUnpaid.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 *deleteSomeUnpaidTickets* method through JDBC, as described in the previous sectio). You should include the *deleteUnpaid.pgsql* source file in the zip file of your submission, along with your versions of the Java source files *FlyingApplication.java* and *RunFlyingApplication.java* that were described in Section 4.

A guide for defining stored procedures/functions with PostgreSQL can be found here on the PostgreSQL site. Note that PostgreSQL stored procedures/functions have some syntactic differences from the PSM stored procedures/functions described in class.

## 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**. It 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.

*RunFlyingApplication.java* is the program that you will need to modify in ways that are similar to the content of the *RunStoreApplication.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 *getAirlinesWithManyFlights* method with the *numberOfFlights* argument set to 3. Your code should print the results returned. Remember that your method should run correctly for any value of *numberOfFlights*, not just when it's 3.

  You should also include the output you got with the *numberOfFlights* argument set to 3 in a comment directly included in the Java source of *RunFlyingApplication*. The comment should be as follows:

  ```
  /*
  *Output of getAirlinesWithManyFlights
  *when the parameter numberOfFlights is 3.
  output here
  */
  ```

- Write one test for the *raiseAirlineTicketCosts* method that raises the cost of flights on AirlineID 'UAL' by 25.

- Write two tests for the *deleteSomeUnpaidTickets* method. The first test should have deleteCount value 1. The second test should have deleteCount value 2. Your code should print the result (total cost of deleted tickets) returned by each test. Note that you need to run the tests in this order, running with deleteCount 1 and then with deleteCount 2. The order affects your results.

**Important:** You must run all of these method tests in order on the original database provided by our create and load scripts. Some of these methods change the database, so using original database and executing methods in order matters.

## 7. Submitting

1. Remember to add comments to your Java code so that the intent is clear.
2. Place the java programs FlyingApplication.java and RunFlyingApplication.java, and the stored procedure declaration code *deleteUnpaid.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 FlyingApplication.java RunFlyingApplication.java deleteUnpaid.pgsql*
4. Lab4 is due on Canvas by 11:59pm on Sunday, December 3, 2017. Late submissions will not be accepted, and there will be no make-up Lab assignments.