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_load_data.sql. The lab4_create.sql script creates all tables within the schema Lab4; otherwise, it is the same as the create.sql in our solution to Lab1, except that the status attribute in OrderSupply is now CHAR(20), instead of CHAR(4). Lab3's new General constraints and revised Foreign Key constraints are not in this schema. The file 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 without qualifying them with the schema name:

```
ALTER ROLE cse182 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 installing Python and psycopg2 in your Docker container

The first step is to install the Python language and the library psycopg2 inside your Docker container. After firing up the container, you can achieve this by executing the following the following three lines from your local environment (i.e., outside the docker container):

docker exec container-psql apt-get update
docker exec container-psql apt-get install -y python3
docker exec container-psql apt-get install python3-psycopg2

3. Instructions for database access from Python

The *runPharmacyApplication.py* which you've been given under Resources→Lab4 is not executable as is. You will have to complete it to make it runnable, and that includes writing the three Python 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 Python functions; that Stored Function is described in Section 5 of this document. We assume that CSE 182 students are familiar with Python 3. *runPharmacyApplication.py* will be the only file in your Python program.

Assuming that you are using the Docker container as usual, and *runPharmacyApplication.py* is in your current directory, you can execute it with the following command:

docker exec container-psql python3 runPharmacyApplication.py

Note: If you have been using a different userid and password than "cse182" and "database4me" for PostgreSQL, then you can specify them as arguments as follows:

docker exec container-psql python3 runPharmacyApplication.py <your_userid> <your_password>

4. 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, both executing SQL statements that SELECT and UPDATE, and executing a Stored Function.

5. Description of the three Python functions in the runPharmacyApplication.py file that interact with the database

The *runDentalApplication.py* file that you've been given contains skeletons for three Python functions that interact with the database using psycopg2. These three Python functions are described below. The first argument for all of these Python functions is your connection to the database. Your task is to implement functions which match those descriptions.

The following psycopg2-related links appear in Lecture 11, which describes how to access databases from Python. All of these links should be helpful, but none of them are perfect introductory tutorials.

- Python PostgreSQL Tutorial Using psycopg2 on the PYnative site, showing some SELECT, INSERT, DELETE and UPDATE statement. This one uses Python 3, which is why it's the first link on the list.
- psycopg2 Tutorial on the PostgreSQL site. Unfortunately, code is written in Python 2, not Python 3, but it explains use of psycopg2 pretty well.
- psychopg2 usage examples, on the psycopg site. Once again, code is written in Python 2, not Python 3, but it explains use of psycopg2.

- *countNumberOfCustomers:* The Purchase table tells us about purchases made by a customer (customerID) at a pharmacy (pharmacyID). Besides the database connection, the countNumberOfCustomers function has one parameter, an integer, thePharmacyID.

  We want to count the number of different customers that a particular pharmacy has had. The function countNumberOfCustomers should count the number of different customers who have made a purchase at the pharmacy identified by the parameter thePharmacyID. The function should return that count.

  But it's possible that there isn't a pharmacy whose pharmacyID value equals thePharmacyID. In that case, countNumberOfCustomers should be -1. And it is possible that there is a pharmacy whose pharmacyID value equals thePharmacyID, but that pharmacy has no customers. In that case, countNumberOfCustomers should return 0 (since the pharmacy has 0 different customers).

  Note that Python functions can have more than one SQL statement in them. You will probably have more than one SQL statement in your code for *countNumberOfCustomers*.

- *updateOrderStatus*: In the OrderSupply table, the value of status is either 'dlvd' (delivered), 'pndg' (pending), or 'cnld' (cancelled). These are the only status values in the Lab4 load data. Besides the database connection, the function *updateOrderStatus* has another parameter, currentYear, which is an integer. The function *updateOrderStatus* should do the following:

  a) If currentYear is between 2000 and 2030 (inclusive), then the status of orders that are delivered or pending should have the string ‘ AS OF <currentYear>’ appended at the end of its status (without the <> characters). For example, assuming that currentYear is 2025 and the status of a tuple is ‘pndg’, then it should be changed to ‘pndg AS OF 2025’. No changes should be made to orders with status ‘cnld’.
  b) If currentYear is less than 2000 or greater than 2030, then no changes should be made to any status values, and *updateOrderStatus* should return -1.

  But how can you concatenate strings together in SQL? As the Lecture 6 slide whose title is "UPDATE with Subquery" illustrated, you employ the operator || (which is the vertical OR bar written twice) to concatenate two strings together. That slide used:

  ```
  'Pres. ' || execName
  ```

  to append 'Pres. ' to the beginning of an execName.

- *deleteSomeOrders*: Besides the database connection, this Python function has one other parameters, maxOrderDeletions, which is an integer.

  deleteSomeOrders invokes a Stored Function, *deleteSomeOrdersFunction*, that you will need to implement and store in the database according to the description in Section 5. The Stored Function *deleteSomeOrdersFunction* has the same maxOrderDeletions parameter as *deleteSomeOrders* (but the database connection is not a parameter for the Stored Function), and *deleteSomeOrdersFunction* returns an integer.

  Section 5 tells you which orders to delete, and explains the integer value that *deleteSomeOrdersFunction* returns. The *deleteSomeOrders* Python function returns the same integer value that the *deleteSomeOrdersFunction* Stored Function returns.

  deleteSomeOrdersFunction doesn't print anything. The *deleteSomeOrders* function must only invoke the Stored Function *deleteSomeOrders*, which does all of the work for this part of the assignment; *deleteSomeOrders* should not do any of the work itself.

6. Stored Function

As Section 4 mentioned, you should write a Stored Function (not a Stored Procedure) called *deleteSomeOrdersFunction* that has a parameter identifying the maximum number of orders it should delete (maxOrderDeletions). The database connection is not a parameter for the Stored Function.

If maxOrderDeletions is less than or equal to 0, *deleteSomeOrdersFunction* should return the value -1. Otherwise, proceed as described below.

Let's refer to orders whose date is 2024-01-05 or earlier as "past orders", and orders whose date is after 2024-01-05 as "future orders". *deleteSomeOrdersFunction* will only delete some future orders. You should know how to compute the number of future orders for each supplier.

The OrderSupply table has an attribute status. For each supplier, we can count the number of "past order" tuples for that supplier in OrderSupply for which status is ‘cnld’. We'll call that "the number of cancelled orders" for the supplier.

The stored function will delete all the "future orders" for some suppliers who have a large "number of cancelled orders". If our function deletes future orders for some supplier, it will delete all of that supplier's future orders. The number of future orders it will delete can't be more than maxOrderDeletions, but it might be less than maxOrderDeletions.

Suppose that the following five suppliers have cancelled orders, and no other suppliers have cancelled orders.

supplierName	number of cancelled orders	number of future orders
McKesson	5	3
AmerisourceBergen	4	6
Cardinal Health	4	2
Cencora	3	1
Medline	2	5

This table is in decreasing order of number of cancelled orders. When two suppliers have the same number of cancelled orders (AmerisourceBergen and Cardinal Health, in this example), they appear in alphabetical order based on supplierName. (If there are equal supplierName values, the order in which they appear doesn't matter.)

The stored function will delete all future orders for some suppliers based on this ordering, keeping track of the total number of orders that it has deleted.

Note: “Cancelled” refers to an order with a status ‘cnld’ in the database (however a tuple still exists for the order). By “delete”, we mean actually removing the a tuple from the database. The function that you will write **deletes** the relevant tuples from the database instead of changing the status of the order.

Let's discuss some examples of the orders that *deleteSomeOrdersFunction* should delete based on the example above.

- If maxOrderDeletions is 20, then *deleteSomeOrdersFunction* deletes all 17 (3 + 6 + 2 + 1 + 5) of the future orders for suppliers with cancelled orders, and returns the number of future orders that it cancelled, which is 17.

- If maxOrderDeletions is 16, then *deleteSomeOrdersFunction* deletes 12 future orders (3 + 6 + 2 + 1) for suppliers with cancelled orders. It doesn't delete any of the future orders for Medline, since that would make the total number of future orders deleted be 17, which is more than 16. The function returns the number of future orders deleted, which is 12.

- If maxOrderDeletions is 10, then *deleteSomeOrdersFunction* deletes 9 future orders (3 + 6) of the future orders for suppliers with cancelled orders. Even though AmerisourceBergen and Cardinal Health have the same number of cancelled orders, AmerisourceBergen's future orders are considered for deletion before Cardinal Health's future orders. And Cardinal Health's future orders can't be deleted, since that would make the total number of future orders deleted be 11, which is more than 10. Moreover, as soon as the function encounters a supplier (Cardinal Health) whose future orders it can't delete, it returns, so Cencora's 1 future order isn't deleted. The function returns the number of future orders that it deleted, which is 9.

- If maxOrderDeletions is 3, then *deleteSomeOrdersFunction* deletes 3 future orders for suppliers with cancelled orders, just the future orders for Smith. The function returns the number of future orders that it deleted, which is 3.

- If maxOrderDeletions is 2, then *deleteSomeOrdersFunction* cannot delete the 3 future orders for Smith, so it does not consider any other suppliers. The function returns the number of future orders that it deleted, which is 0.

- If maxOrderDeletions is 3, then *deleteSomeOrdersFunction* deletes 3 future orders of the future orders for suppliers with cancelled orders, just the future orders for McKesson. The function returns the number of future orders that it deleted, which is 3.

- If maxOrderDeletions is 0, that's an error, and *deleteSomeOrdersFunction* returns -1.

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

```
\i deleteSomeOrdersFunction.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 *cancelSomeOrders* function in your Python 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 cancelSomeOrdersFunction.pgsql source file in the zip file of your Submission, together with your versions of the Python file runPharmacyApplication.py that was described in Section 4. See Section 7 for detailed Submission 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 IN parameters.

We've given you some more hints in Lecture 11 and in the Lab4 announcement 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*.

7. Testing

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

- Write four tests in *runPharmacyApplication.py* of the Python function *countNumberOfCustomers*.
  - The first test should be thePharmacyID value 11.
  - The second test should be for thePharmacyID value 17.
  - The third test should be for thePharmacyID value 44.
  - The fourth test should be for thePharmacyID value 66.

  1) If a test of *countNumberOfCustomers* returns a value greater or equal to zero, then print the following message:

     Number of customers for pharmacy <thePharmacyID> is <number of customers>

     where < thePharmacyID> is the parameter value provided, and
     <number of customers > is the value returned by *countNumberOfCustomers*.

  2) However, if a test of *countNumberOfCustomers* returns a negative value, then you should print out the values of its parameter, with a message explaining the error that occurred. You may choose the format yourself, as long as the error explanation is clear. You should continue executing further tests, even if *countNumberOfCustomers* returns a negative value.

     In both cases 1) and 2), the output should be followed by an extra blank line.

- Write three tests in *runPharmacyApplication.py* of the Python function *updateOrderStatus*.
  - The first test should be for currentYear value 1999.
  - The second test should be for currentYear value 2025.
  - The third test should be for currentYear value 2031.

  1) If a test of *updateOrderStatus* returns a negative value, then you should print out the values of its parameter, with a message explaining the error that occurred. You may choose the format yourself, as long as the error explanation is clear. You should continue executing further tests, even if *updateOrderStatus* returns a negative value.

  2) But if a test of *updateOrderStatus* returns a non-negative value, then you should print out its result (which is the number of OrderSupply tuples that were updated) using the following format:

  Number of orders whose status values were updated by updateOrderStatus is <number of status values updated>

  where <number of status values updated> is the number of status values that were updated.

  [The output for each invocation of *updateOrderStatus* should appear on a single line, not split into two lines, and you should print a blank line after each.]

  In both cases 1) and 2), the output should be followed by an extra blank line.

- Also write four tests in *runPharmacyApplication.py* of the Python function *deleteSomeOrders*.
    - The first test should be for maxOrderDeletions 2.
    - The second test should be for maxOrderDeletions 4.
    - The third test should be for maxOrderDeletions 3.
    - The fourth test should be for maxOrderDeletions 1.

  1) If a test of *deleteSomeOrders* returns a non-negative value, then you should print out its result (which is the number of orders that were deleted) using the following format:

  Number of orders which were deleted for maxOrderDeletions value <maxOrderDeletions> is <number of orders that were deleted>

  where maxOrderDeletions is the value of the maxOrderDeletions parameter provided, and <number of orders that were deleted> is the number of orders in OrderSupply that were deleted.

  [The output for each invocation of *deleteSomeOrders* should appear on a single line, not split into two lines, and you should print a blank line after each.]

  2) But if a test of *deleteSomeOrders* returns a negative value, then you should print out the values of its parameter, with a message explaining the error that occurred, followed by a blank line. You may choose the format yourself, as long as the error explanation is clear. You should continue executing further tests, even if *deleteSomeOrders* returns a negative value.

You must run all of these function tests in order, starting with the database provided by our create and load scripts. Some of these functions change the database, so using the load data that we've provided and executing the functions in order is required. You should not reload the data multiple times in Lab4.

Does the order in which these tests are run matter? What do you think?

8. Submitting

1. Remember to add comments to your Python code so that the intent is clear.
2. Save the Python program runPharmacyApplication.py and the stored procedure declaration code *deleteSomeOrdersFunction.pgsql* in your working directory.
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 runPharmacyApplication.py deleteSomeOrdersFunction.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 might want to use views to do Lab4. That's not required, but it is permitted. If you do use views, you must put the statements creating those views in a file called createPharmacyViews.sql, and include that file in your Lab4 zip file.
5. Lab4 is due on Canvas by 11:59pm on **Tuesday, June 3**. Late submissions will not be accepted, and there will be no make-up Lab assignments.