1 Preliminaries

In this lab, you will work with a database schema similar to the schema that you used in Lab2. We have provided a create_lab3.sql script for you to use (which is similar to, but not quite the same as the create.sql in our Lab2 solution), so that everyone can start from the same place. Please remember to DROP and CREATE the Lab3 schema before running that script (as you did in previous labs), and also execute:

```
ALTER ROLE cse182 SET SEARCH_PATH TO Lab3;
```

so that you'll always be using the Lab3 schema without having to mention it whenever you refer to a table.

You will need to log out and log back in to the server for this default schema change to take effect. (Students often forget to do this.)

We have provided a load_lab3.sql script that will load data into your tables. You will need to run that script before executing Lab3. The command to execute a script is: \i <filename>

In Lab3, you will be required to combine new data (as explained below) into one of the tables. You will need to add some new constraints to the database and do some unit testing to see that the constraints are followed. You will also create and query a view, and create an index.

New goals for Lab3:

1. Perform SQL to "combine data" from two tables
2. Add foreign key constraints
3. Add general constraints
4. Write unit tests for constraints
5. Create and query a view
6. Create an index

There are lots of parts in this assignment, but none of them should be too difficult. Lab3 will be discussed during the Lab Sections before the due date, Tuesday, May 20.

2. Description

2.1 Tables with Primary Keys for Lab3

The primary key for each table is underlined. The attributes are the same ones from Lab2, but there's one table that you haven't seen before.

Customer(customerID, customerName, address, dateOfBirth, healthInsuranceName)

Pharmacy(pharmacyID, address, openTime, closeTime, numEmployees)

Drug(drugID, drugName, manufacturer, prescriptionRequired)

Supplier(supplierID, supplierName, rating)

Purchase(purchaseID, customerID, pharmacyID, purchaseTimestamp, totalPrice, creditCardType, creditCardNumber, expirationDate)

DrugsInPurchase(purchaseID, drugID, quantity, subtotal, discount)

OrderSupply(pharmacyID, supplierID, drugID, drugPrice, quantity, orderDate, status)

UpgradeSupplier(supplierID, supplierName)

In the create_lab3.sql file that we've provided under Resources→Lab3, the first 7 tables are similar to the tables were in our Lab1 create_lab1.sql solution; the NULL and UNIQUE constraints from Lab2 are **not** included. Moreover, create_lab3.sql is missing the Foreign Key Constraints on Visits (which say that the visiting patient must be a patient in Patients, and that the visited dentist must be a in Dentists), as well as a Foreign Key Constraint on TreatmentsDuringVisits (which says that a visit in TreatmentsDuringVisits must be a visit in the Visits table) that were in create_lab1.sql. You'll create new variations of those constraints in Section 2.3 of Lab3.

In practice, primary keys, unique constraints, and other constraints are almost always entered when tables are created, not added later. create_lab3.sql handles some constraints for you, but, you will be adding some additional constraints to these tables in Lab3, as described below.

Note also that there is an additional table, UpgradeSupplier, in the create_lab3.sql file that has most (but not all) of the attributes that are in the Supplier table. We'll say more about UpgradeSupplier below.

Under Resources→Lab3, we've provided a load script named lab3_load_data that loads tuples into the tables of the schema. **You must run both create_lab3.sql and lab3_load_data.sql before you run the parts of Lab3 that are described below.**

2.2 Combine Data

Write a file, *combine.sql* (which should have multiple SQL statements that are in a Serializable transaction) that will do the following. For each tuple in UpgradeSupplier, there might already be a tuple in the Supplier table that has the same Primary Key (that is, the same value for supplierID). If there **isn't** a tuple in Supplier with the same Primary Key, then this is a new supplier which should be inserted into the Supplier table. If there **already is** a tuple in Supplier with that Primary Key, then this is an update of information about that supplier. So here are the effects that your transaction should have:

- If there isn't already a tuple in the Supplier table which has that supplierID, then you should insert a tuple into the Supplier table which has the supplierID and supplierName value that are in the UpgradeSupplier tuple. Also, the rating for that inserted tuple should be 9.

- If there already is a tuple in the Supplier table which has that supplierID, then update the tuple in Supplier which has that supplierID using the supplierName value in the UpgradeSupplier tuple. Also, increase the rating for that existing tuple by 1.

Your transaction may have multiple statements in it. The SQL constructs that we've already discussed in class are sufficient for you to do this part (which is one of the hardest parts of Lab3).

2.3 Add Foreign Key Constraints

Important: Before running Sections 2.3, 2.4 and 2.5, recreate the Lab3 schema using the *create_lab3.sql* script, and load the data using the script *load_lab3.sql*. That way, any database changes that you've done for Combine won't propagate to these other parts of Lab3.

Here's a description of the Foreign Keys that you need to add for this assignment. (Foreign Key Constraints are also referred to as Referential Integrity constraints.) The create_lab3.sql file that we've provided for Lab3 includes only some of the Referential Integrity constraints that were in the Lab2 solution, but you're asked to use ALTER to add additional constraints to the Lab3 schema.

The load data that you've been given should not cause any errors when you add these constraints. Just add the constraints listed below, exactly as described, even if you think that additional Referential Integrity constraints should exist. Note that (for example) when we say that each customer (customerID) that appears in the Purchase table must appear in the Customer table, that means that the customerID attribute of the Purchase table is a Foreign Key referring to the Primary Key of the Customer table (which also is customerID).

- Each customer (customerID) that appears in the **Purchase** table must appear in the Customer table as a Primary Key (customerID). If a tuple in the Customer table is deleted, then all Puchase tuples whose customerID equals that customerID should also be deleted. If the Primary Key (customerID) of a tuple in the Customer table is updated, then all Purchase tuples for that updated customer should also be updated to the new customerID value.

- Each pharmacy (pharmacyID) that appears in the **Purchase** table must appear in the Pharmacy table as a Primary Key (pharmacyID). If a tuple in the Pharmacy table is deleted and there are Purchase tuples which correspond to that pharmacyID, then that pharmacy tuple deletion should be rejected. If the Primary Key (pharmacyID) of a tuple in the Pharmacy table is updated, then all Purchase tuples for that updated pharmacy should also be updated to the new pharmacyID value.

- Each purchase (purchaseID) that appears in the **DrugsInPurchase** table must appear in the Purchase table as a Primary Key (purchaseID). If a tuple in the Purchase table is deleted and there are DrugsInPurchase tuples that correspond to that purchase, then all DrugsInPurchase tuples which correspond to the deleted purchase should also be deleted. If a Primary Key in the Purchase table is updated, then all DrugsInPurchase tuples which correspond to that purchase's Primary Key should also be updated the same way.

Write commands to add foreign key constraints in the same order that the foreign keys are described above. Your foreign key constraints should all have names, but you may choose any names that you like. Save your commands to the file *foreign.sql*

2.4 Add General Constraints

The general constraints for Lab3, which should be written as CHECK constraints, are:

1. In Pharmacy, numEmployees must be greater than zero. This constraint should be named pharmacyEmployeesPositive.

2. In OrderSupply, the value of status must be 'dlvd', 'pndg', 'cnld', or NULL. This constraint should be named validOrderStatus.

3. In Purchase, if creditCardType is NULL, then creditCardNumber must also be NULL. This constraint should be named ifNullTypeThenNullNumber.

Write commands to add general constraints in the order the constraints are described above, and save your commands to the file *general.sq*l. Remember that values TRUE and UNKNOWN are okay for a CHECK constraint, but FALSE is not.

2.5 Write Unit Tests

Unit tests are important for verifying that your constraints are working as you expect. We will require tests for just a few common cases, but there are many more unit tests that are possible. The unit tests you write need to do what's described below on the specific database instance that contains the Lab3 load data.

For each of the 3 foreign key constraints specified in section 2.3, write one unit test:

- An INSERT command that violates the foreign key constraint (and elicits an error). You must violate that specific foreign key constraint, not any other constraint.

Also, for each of the 3 general constraints, write 2 unit tests. This means that you will write 2 tests for the first general constraint, followed by 2 tests for the second general constraint, followed by 2 tests for the third general constraint.

- An UPDATE command that meets the constraint.
- An UPDATE command that violates the constraint (and elicits an error).

Save these 3 + 6 = 9 unit tests in the order specified above in the file *unittests.sql*.

2.6 Working with a View

Important: Before starting Section 2.6, recreate the Lab3 schema once again using the *create_lab3.sql* script, and load the data using the script *load_lab3.sql*. That way, any changes that you've done for previous parts of Lab3 (e.g., Unit Test) won't affect the results of this query.

2.6.1 Create a view

A purchase in the Purchase table has a totalPrice paid by the customer. But each drug in a purchase (in the DrugsInPurchase table) also has a price given by the subtotal attribute, and a single purchase may contain multiple drugs. The total price of drugs that a customer pays (which we'll call "total drug price") may not be the same as the total price of that customer's purchase (totalPrice, which is an attribute in the Purchase table).

We're going to find major inconsistencies between totalPrice for a purchase and the "drug price", which can be computed by adding up all the subtotal values for DrugsInPurchase for that purchase. (Note that the quantity attribute should be ignored here as it is already taken into account in the subtotal attribute.)

Create a view called BadTotalPriceView, which identifies the purchases where the difference between totalPrice and "total drug price" is greater than or equal to 1. The attributes in the view should be the attributes in the Purchase table (namely purchaseID, customerID, pharmacyID, purchaseTimestamp, totalPrice, creditCardType, creditCardNumber, and expirationDate) plus an attribute which we'll call totalDrugPrice, which is the "total drug price" for that purchase. But only include a tuple in BadTotalPriceView if the difference between totalPrice and "total drug price" is greater than or equal to 1. (Note that either price value could be larger than the other.)

No duplicates should appear in your view.

Save the script for creating that view in a file called *createview.sql*

2.6.2 Query a View

For this part of Lab3, you'll write a script called *queryview.sql* that contains a query which we'll informally refer to as the MajorPriceDiscrepancy query. (You don't actually need to name the query, or create another view for this.) The MajorPriceDiscrepancy query uses **BadTotalPriceView**, and (possibly) some other tables. In addition to writing the MajorPriceDiscrepancy query, you must also include some comments in the queryview.sql script; we'll describe those necessary comments below.

For tuples in the result of **BadTotalPriceView**, we can compute the value of totalPrice minus totalDrugPrice; let's call that value (which doesn't appear in **BadTotalPriceView**) priceDifference.

Write and run a SQL query which finds the total priceDifference for each pharmacy that appears in **BadTotalPriceView**, as well as the number of tuples for that pharmacy that are in **BadTotalPriceView**. The attributes in your result should be pharmacyID, address, totalPriceDifference, and badTotalPriceCount.

Then write the results of the MajorPriceDiscrepancy query in a comment. *The format of that comment is not important; the comment just has to have all the right information in it.*

Next, write a SQL statement that deletes the following tuple from the DrugsInPurchase table:

- The tuple whose Primary Key is (7, 10).

Run the MajorPriceDiscrepancy query once again after that update. Write the output of the query in a second comment. Do you get a different answer?

You need to submit a script named *queryview.sql* containing your query on the views. In that file you must include:

1) Your MajorPriceDiscrepancy SQL query.
2) A comment with the output of that query on the load data before the update.
3) A SQL statement which performs the update described above.
4) Repeat your MajorPriceDiscrepancy SQL query.
5) A second comment with the output of that query after the update.

It probably was a lot easier to write the MajorPriceDiscrepancy query using the view than it would have been if you didn't have the view!

2.7 Create an Index

Indexes are data structures used by the database to improve query performance. Locating the tuples in the Purchase table for a particular creditCardType and totalPrice might be slow, if the database system has to search the entire Purchase table (if its size was very large). To speed up that search, create an index named PurchaseIndex over the creditCardType and totalPrice columns (in that order) of the Purchase table. Save the command in the file *createindex.sql*.

Of course, you can run the same SQL statements whether or not this index exists; having indexes just changes the performance of SQL statements.

For this assignment, you need not do any searches that use the index, but if you're interested, you might want to do searches with and without the index, and look at query plans using EXPLAIN to see how queries are executed. Please refer to the documentation of PostgreSQL on EXPLAIN that's at https://www.postgresql.org/docs/15/sql-explain.html

3 Testing

Before you submit, login to your database via psql and execute the provided database creation and load scripts, and then test your seven scripts (combine.sql foreign.sql general.sql unittests.sql createview.sql queryview.sql createindex.sql). Note that there are two sections in this document (both labeled **Important**) where you are told to recreate the schema and reload the data before running that section, so that updates you performed earlier won't affect that section. Please be sure that you follow these directions, since your answers may be incorrect if you don't.

4 Submitting

1. Save your scripts indicated above as combine.sql foreign.sql general.sql unittests.sql createview.sql queryview.sql createindex.sql. You may add informative comments inside your scripts if you want (the server interprets lines that start with two hyphens as comment lines).

2. Zip the files to a single file with name Lab3_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 Lab3 should be named Lab3_1234567.zip To create the zip file you can use the Unix command:

   zip Lab3_1234567 combine.sql foreign.sql general.sql unittests.sql createview.sql queryview.sql createindex.sql

   (Of course, you use your own student ID, not 1234567.)

3. Lab3 is due on Canvas by 11:59pm on Tuesday, May 20. Late submissions will not be accepted, and there will be no make-up Lab assignments.