1 Preliminaries

In this lab, you will work with a Restaurant database schema similar to the schema that you used in Lab2. We've provided a lab3_create.sql script for you to use (which is 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 yourlogin 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're also providing a lab3_data_loading.sql script that will load data into your tables. You'll need to run that script before executing Lab3. The command to execute a script is: \i <filename>

You will be required to combine new data (as explained below) into one of the tables. You will also 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 difficult. Lab3 will be discussed during the Lab Sections before the due date, Tuesday, November 19. (You have nearly an extra week to do this Lab because of the Midterm.) But note that Monday, November 11 is a holiday, Presidents Day, so there won't be a class or Lab Section on that day.

2. Description

2.1 Tables with Primary Keys for Lab3

The primary key for each table is underlined.

customer(custID, name, address, joinDate, status)

menuItem(menuItemID, name, description, price)

dinnerTable(dinnerTableID, numSeats, InUse)

server(serverID, name, level, salary)

visit(visitID, custID, dinnerTableID, serverID, numPeople, cost, custArrive, custDepart)

billEntry(visitID, menuItemID, quantity)

modifyVisit(visitID, custID, dinnerTableID, serverID, numPeople)

In the lab3_create.sql file that we've provided under Resources→Lab3, the first 6 tables are the same as they were in our Lab2 solution, including NULL and UNIQUE constraints, but there are no Referential Integrity constraints on the visit table. (The referential integrity constraints on the billEntry table from Lab2 have been retained.)

In practice, primary keys, unique constraints and other constraints are almost always entered when tables are created, not added later. lab3_create.sql handles some constraints for you, but, you will be adding some additional constraints to these tables in Lab3, as described below.

Note that there is an additional table, modifyVisit that has some (but not all) of the attributes as the visit table. We'll say more about modifyVisit below.

Under Resources→Lab3, you'll also be given a load script named lab3_data_loading.sql that loads tuples into the tables of the schema. You must run both lab3_create.sql and lab3_data_loading.sql before you run the parts of Lab3 that are described below.

2.2 Combine Data

Write a file, *combine.sq*l (which should have multiple SQL statements in it) that will do the following. For each "new customer" tuple in modifyVisit, there might already be a tuple in the visit table that has the same visitID. If there **isn't** a tuple with the same visitID , then this is a new visit that should be inserted. If there already **is** a tuple with that visitID, then this is an update of information about that visit tuple. So here are the actions that you should take.

- If there **isn't** already a tuple in the visit table that has that visitID, then insert a tuple into the visit table corresponding to that modifyVisit tuple. Use visitID, custID, dinnerTableID, serverID and numPeople, as provided in the modifyVisit tuple. Set custArrive to be the current time; we explain how you do in the Lab3 announcement on Piazza. Set cost and custDepart to NULL.

- If there already **is** a tuple in the visit table that has that visitID, then update the visit table based on that modifyVisit tuple. Don't change custArrive, custDepart or cost for that visit, but update custID, dinnerTableID, serverID and numPeople based on the values of those attributes in the modifyVisit tuple. (There may have be a typo when that information was originally entered.)

You may be able to do this with a single SQL statement, or you may use multiple SQL statements to do this. In practice, statement would be executed in a transaction, but we're not requiring that for Lab3, because we haven't discussed transactions yet. A helpful hint is provided in the initial Lab3 announcement posted on Piazza.

2.3 Add Foreign Key Constraints

Important: Before running Sections 2.3, 2.4 and 2.5, recreate the Lab3 schema using the *lab3_create.sql* script, and load the data using the script *lab3_data_loading.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. Note that although there were Referential Integrity constraints in the Lab2 solution, there are no Referential Integrity constraints in the create.sql file for Lab3.

The load data that you're provided with should not cause any errors. Just add the constraints listed below in the form listed, even if you think that different Referential Integrity constraints should exist. Note that (for example) when we say that a customer in the visit table must appear in the customer table, that refers to the custID field in both tables.

- Each dinnerTableID that's in the visit table must appear in the dinnerTable table. If there are visits for a particular dinner table, then updates or deletions of the corresponding dinnerTableID attribute tuple in the dinnerTable table should be rejected.
- Each custID that's in the visit table must appear in the customer table. If a tuple in the customer table is deleted, then all visits for the corresponding custID should be deleted. If there are visits for a particular custID, then any update of the custID in the customer table should be rejected.
- Each serverID that's in the visit table must appear in the server table. If the serverID for a tuple in the server table is updated, then all visits for the corresponding serverID should be updated the same way . If a tuple in the serverID table is deleted, then any tuples in the visits table corresponding to that serverID should have their serverID field set to NULL.

Write commands to add foreign key constraints in the same order that the foreign keys are described above. Save your commands to the file *foreign.sql*

2.4 Add General Constraints

General constraints for Lab3 are:

1.In menuItem, price must be positive. Please give a name to this constraint when you create it. We recommend that you use the name positive_price, but you may use another name. The other general constraints don't need names.

2. In visit, custArrive must be less than or equal to custDepart.

3. In dinnerTable, if inUse is not NULL, then numSeats must greater than 0.

Write commands to add general constraints in the order the constraints are described above, and save your commands to the file *general.sq*l. (Note that UNKNOWN for a Check constraint is okay, but FALSE isn't.)

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.

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).

Also, for each of the 3 general constraints, write 2 unit tests:

- 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 given above, in the file *unittests.sql*.

2.6 Working with views

2.6.1 Create a view

Although the visit table has a cost field, there's another way that we can calculate the cost of a visit. For each visitID, there may be billEntry tuples for that visitID. Each billEntry tuple has a quantity, and each billEntry tuple has a menuItemID that identifies a menuItem tuple that has a price. So the calculatedCost of a visit can be calculated by adding up price*quantity for all of the billEntry tuples for that visit.

Create a view called costView that has 2 attributes, visitID and calculatedCost This view should have a tuple for each visitID that gives the calculated cost for that visitID. Your view should have no duplicates in it.

And as you've probably already deduced, you'll need to use a GROUP BY in your view. But there's one challenging aspect of this problem: What happens if there's a visitID for which there are no billEntry tuples? Well, there still should be a tuple for that visitID in the costView, and that tuple's calculatedCost should be 0.

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

2.6.2 Query view

Write and run a SQL query over the costView view to answer the following "Incorrect Costs for Frequent Customers" question. You'll to use some tables to write this query, but be sure to use the view.

> Let's define a Frequent Customer to be a customer who visited at least 3 times. For some visits, the cost that appears in the visit table is not the same as the calculatedCost in the costView view. You should output the visitID, customer name, cost and calculatedCost for each visit that was made by a Frequent Customer in which the cost and calculatedCost are different. In your result, the attributes should appear as visitID, customerName, cost and calculatedCost. No duplicates should appear in your result.

Important: Before running this query, recreate the Lab3 schema once again using the *lab3_create.sql* script, and load the data using the script *lab3_data_loading.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. Then write the results of that query in a comment.

Next, write commands that delete just the tuples that have the following primary keys from the billEntry table:

- the tuple with visitID 10, menuItem 3, and
- the tuple with visitID 2, menuItem 1.

Run the "Incorrect Costs for Frequent Customers" query once again after those deletions. 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 also include:

- the comment with the output of the query on the provided data before the deletions,
- the SQL statements that delete the tuples indicated above,
- and a second comment with the second output of the same query after the deletions.

You do not need to replicate the query twice in the *queryview.sql* file (but you won't be penalized if you do).

It probably was easier to write this query using the costView view than it would have been without that view.

2.7 Create an index

Indexes are data structures used by the database to improve query performance. Locating all of tuples in the billEntry table for a particular menuItem might be slow if the database system has to search the entire billEntry table. To speed up that search, create an index named LookUpBillItems over the menuItemID and quantity columns (in that order) of the billEntry 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. But there index could make it faster to search for the billEntry entries in which a particular menuItem had been ordered with a quantity that was 3 or more.

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/10/static/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. You should already know how to transfer the files from the UNIX timeshare to your local machine before submitting to Canvas.
4. Lab3 is due on Canvas by 11:59pm on Tuesday, November 19. Late submissions will not be accepted, and there will be no make-up Lab assignments.