1 Preliminaries

In this lab, you will work with a Gavotte movie theater 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 similar to but 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 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'll also provide 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, Sunday, February 23. (You have nearly an extra week to do this Lab because of the Midterm.) But note that Monday, February 17 is a holiday, Presidents Day, so there won't be a Lecture on that day.

2. Description

2.1 Tables with Primary Keys for Lab3

The primary key for each table is underlined.

Movies(movieID, name, year, rating, length, totalEarned)

Theaters(theaterID, address, numSeats)

TheaterSeats(theaterID, seatNum, brokenSeat)

Showings(theaterID, showingDate, startTime, movieID, priceCode)

Customers(customerID, name, address, joinDate, status)

Tickets(theaterID, seatNum, showingDate, startTime, customerID, ticketPrice)

ModifyShowings(theaterID, showingDate, startTime, movieID)

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 Tickets table**. (That's one difference between the lab3_create.sql file and the create.sql file in our Lab2 solution; the ModifyShowings tables is another.) Referential integrity constraints on the other tables 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 also that there is an additional table, ModifyShowings, that has most (but not all) of the attributes in the Showings table. We'll say more about ModifyShowings 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.sql* (which should have multiple SQL statements that are in a Serializable transaction) that will do the following. For each "new showing" tuple in ModifyShowings, there might already be a tuple in the Showings table that has the same primary key (that is, the same values for theaterID, showingDate and startTime). If there **isn't** a tuple in Showings with the same primary key, then this is a new showing that should be inserted into Showings. If there already **is** a showing tuple in Showings with that primary key, then this is an update of information about that showing. So here are the actions that you should take.

- If there **isn't** already a tuple in the Showings table that has that primary key, then insert a tuple into the Showings table corresponding to that ModifyShowings tuple. Use theaterID, showingDate, startTime, and movieID, as provided in the ModifyShowings tuple. Set priceCode to NULL.

- If there already **is** a showing tuple in the Showings table that has that primary key, then update the tuple based in Showings that has the same primary key as the ModifyShowings tuple. Update movieID for that existing showing tuple based on the value of movieID in the ModifyShowings tuple. Don't change any of the other attributes in that showing tuple.

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). 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 showing in the Tickets table must appear in the Showings table, that says that the Tickets table has a Foreign Key referring to the Primary Key of the Showings table.

- Each showing in the Tickets table must appear in the Showings table. (Explanation of what that means appear in the above paragraph.) If a Showings tuple is deleted and there are tickets for that showing, the deletion should be rejected. Also, if the Primary Key of a Showings tuple is updated, and there are tickets for that showing, then the update should be rejected.
- Each customer that's in the Tickets table must appear in the Customers table. If a tuple in the Customers table is deleted, then all tickets for the corresponding customerID should be have their customerID set to NULL. If customerID for a tuple in the Customers table is updated, then all Tickets tuples for that customerID should be updated the same way.
- Each theater seat in the Tickets table must appear in the TheaterSeats table. [A theater seat is identified by the Primary Key of the TheaterSeats table, (theaterID, seatNum).] If a tuple in the TheaterSeats table is deleted, then all tickets for that theater seat should also be deleted. If the Primary Key of a theater seat in TheaterSeats is updated, then all Tickets for that theater seat in Tickets should be updated the same way.

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

2. In Customers, joinDate must not be before November 27, 2015.

3. In Showings, if movieID is not NULL, then priceCode must also not be NULL.

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 Movies table has a totalEarned field, there's another way that we can calculate the total amount that a movie has earned. For each showing (in Showings) of that movie, there may be ticket tuples (in Tickets) for that movie's movieID. Each Tickets tuple has a ticketPrice. So the computedEarnings of a movie can be calculated by adding up ticketPrice for all of the Tickets tuples that correspond to Showings of that movie.

Create a view called earningsView that has 2 attributes, movieD and computedEarnings This view should have a tuple for each movieID that gives the computedEarnings for that movietID. 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 movieID for which there are no tickets? Well, there still should be a tuple for that movieID in earningsView, and that tuple's computedEarnings should be 0.

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

2.6.2 Query view

Write and run a SQL query over the earningsView view to answer the following "Misreports for Each Rating" question. You may want to use some tables to write this query, but be sure to use the view.

> For some movies, the totalEarnings that appears in the Movies table is not the same as the computedEarnings for that movie in the earningsView view; we'll say that such movies are misreported. For each rating that has at least one misreported movie with that rating, you should output the number of misreported movies with that rating. The attributes in your result should be rating and misreportCount.
>
> However, only include tuples in your result for a rating if all the misreported movies with that rating were made before 2019. 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. *The format of that comment is not important; it just has to have the right information in it.*

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

- the tuple with primary key (111, 1, 2009-06-23, 11:00:00)
- the tuple with primary key (44, 5, 2020-06-24,15:00:00)

Run the "Misreports for Each Rating" 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).

Aren't you glad that you had the earningsView view? It probably was easier to write this query using that view than it would have been if you hadn't had that view. You might even have decided to create yet another view to do this query.

2.7 Create an index

Indexes are data structures used by the database to improve query performance. Locating all of tuples in the Showings table that have a particular showingDate and startTime might be slow if the database system has to search the entire Showings table. To speed up that search, create an index named LookUpShowings over the showingDate and startTime columns (in that order) of the Showings 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 Showings on a particular showingDate whose startTime is after a specific time.

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/11/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 Sunday, February 23. Late submissions will not be accepted, and there will be no make-up Lab assignments.