1 Preliminaries

In this lab, you will work with a Stock Market 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, Wednesday, February 27. (You have an extra week to do this Lab because of the Midterm, which is on Monday, February 11.) But note that Monday, February 18 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.

Exchanges(exchangeID, exchangeName, address)

Stocks(exchangeID, symbol, stockName, address)

Customers(customerID, custName, address, category, isValidCustomer)

Trades(exchangeID, symbol, tradeTS, buyerID, sellerID, price, volume)

Quotes(exchangeID, symbol, quoteTS, price)

NewCustomers(customerID, custName, address)

In the lab3_create.sql file that we've provided under Resources→Lab3, the first 5 tables are the same as they were in our Lab2 solution, including NULL and UNIQUE constraints. Note that there is an additional table, NewCustomers that has some of the same attributes as Customers. As the table name suggests, each of its tuples records a Stock Market customer, which may be either for a new customer, or a modification of an existing customer. We'll say more about NewCustomers below.

In practice, primary keys and unique constraints are almost always entered when tables are created, not added later, and lab3_create.sql handles those constraints for you. However, we will be adding some additional constraints to these tables, as described 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 in it in a Serializable transaction) that will do the following. For each "new customer" tuple in NewCustomers, there might already be a tuple in Customers that has the same primary key, customerID. If there **isn't** a tuple with that customerID, then this is a new Stock Market customer. If there already **is** a tuple with that customerID, then this is an update information about that customer. So here are the actions that you should take.

a) If there **isn't** already a tuple in Customers that has the same primary key, then insert a tuple into the Customers table corresponding to that NewCustomers tuple. Use customerID, customerName and address, as provided in the NewCustomers tuple. Set isValidCustomer to be TRUE; category should be NULL.

b) If there already **is** a tuple in Customers that has the same primary key, then update Customers based on that NewCustomers tuple. Don't change customerID or category or that customer, but update customerName and address based on the values of those attributes in the NewCustomers tuple. (The customer may have changed name and address.) Also, set isValidCustomer to be TRUE.

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. The default for referential integrity should be used in all cases. The load data that you're provided with should not cause any errors. There are other referential integrity constraints that probably would exist for this schema—you might want to think about those--but please just add the constraints listed below.

a) The buyerID field in Trades should reference the customerID primary key Customers.

b) The sellerID field in Trades should reference the customerID primary key in Customers.

c) The (exchangeID, symbol) fields in Trades should reference the corresponding primary key in Stocks.

d) The (exchangeID, symbol) fields in Quotes should reference the corresponding primary key in Stocks.

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 Quotes, price must be positive.

2. In Trades, cost isn't an attribute, but it's price * volume. This constraint says that cost must be positive. Please give a name to this constraint when you create it. We recommend that you use the name positive_cost, but you may use another name. The other general constraints don't need names.

3. In Trades, buyerID and sellerID must be different.

4. In Customers, if category is 'H' then isValidCustomer must be TRUE.

Write commands to add general constraints in the order the constraints are described above, and save your commands to the file *general.sql*. (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 write just a few for the common cases, but there are many more possible tests we could write.

For each of the 4 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 4 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 4 + 8 = 12 unit tests, in the order given above, in the file *unittests.sql*.

2.6 Working with views

2.6.1 Create a view

The primary key for Quotes is (exchangeID, symbol, quoteTS), and each quote in Quotes gives the price of a particular stock (identified by exchangeID and symbol) at the quoteTS TIMESTAMP. You'll create a view that for each date gives the opening price, closing price, low price and high price for stocks. You only have to provide this information for stocks on the days that there are quotes for that stock. For example, if there are 75 quotes for a stock on one date, and there are no quotes for that stock on another date, you'll have a tuple in the view for that stock on the first date, but not for the second date. And as you've probably already deduced, you'll need to use a GROUP BY in your view.

But a TIMESTAMP consists of a DATE and a TIME. How do you extract the DATE from a TIMESTAMP in PostgreSQL? Assume that expressionTS is an expression that evaluates to a TIMESTAMP, which may be a TIMESTAMP constant, a TIMESTAMP attribute, or any other TIMESTAMP expression. Here are two ways to get the date; you may use either for PostgreSQL. But there are variations in type casting in different SQL systems, so these might (or might not) work in other SQL systems.

1- DATE(expressionTS)

2- expressionTS::DATE

You may use these in any clause of a SQL statement in which expressionTS could legally be used.

Create a view named QuotesSummary. For each stock (identified by exchangeID and symbol), there will be a tuple in QuotesSummary for each date on which there was at least one quote for that stock. For each date, the QuotesSummary result should provide the following:

- The opening price for each stock, which is the price for that stock that has the earliest timestamp on that date.
- The closing price for each stock, which is the price for that stock that has the latest timestamp on that date.
- The low price for each stock, which is the lowest price for that stock on that date.
- The high price for each stock, which is the highest price for that stock on that date.

The attributes in your QuotesSummary view should be exchangeID, symbol, theDate, openingPrice, closingPrice, lowPrice and highPrice. Your view should have no duplicates in it.

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

2.6.2 Query view

Write a SQL query over the QuotesSummary view to answer the following "Stocks with Multiple High Closings" question. (You may also have to use some tables to do this, but be sure to use these views.)

> On some dates, a stock closes at its high price for that date. We'll say that a stock has Multiple High Closings if there are at least 2 dates on which the stock's closing price equaled its high price for that date. Write a SQL query that identifies those stocks. For each such stock, the attributes in your result should be the stock's exchangeName (not exchangeID) and stockName (not symbol), and its numHighClosings. numHighClosings should be the number of dates on which the stock closed at its high. Remember: You only want a tuple in your result for stocks where the numHighClosings value is at least 2. Your result should have no duplicates.

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 with the following primary keys from the Quotes table:

(NYSE, CLDR)

(NASDAQ, ANF)

Run the "Stocks with Multiple High Closings" 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 QuotesSummary view than without that view.

2.7 Create an index

Indexes are data structures used by the database to improve query performance. Locating all the Trades made between a particular buyerID and a particular sellerID may be slow if the database system has to search the entire Trades table. To speed up that search, create an index named LookUpTrades over the buyerID and sellerID columns (in that order) of the Trades 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/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 Wednesday, February 27. Late submissions will not be accepted, and there will be no make-up Lab assignments.