Spring 2018 Comp 533
Assignment 5
Due: April 11, 2018 at 11:55 PM

A5: Imperative SQL

The goal of this assignment is to write functions, stored procedures, and triggers that add functionality to our database.

What to turn in

You must turn in a .sql file on Canvas. Basically, we want to be able to hit execute and run your sql code to create your imperative SQL and run it. This means that any comments or text answers in your file should be in SQL comments.

Grading

What's In and Out of Scope

This is intended to be an imperative SQL assignment. Therefore, you must write code, although you may need to embed declarative SQL in some of your code. You may use VIEWs as needed and you may use standard built-in MySQL functions (e.g. ROUND, IF or CASE statements). If you're not sure if something is allowed, ask!

Academic Honesty

The following level of collaboration is allowed on this assignment:

You may discuss the assignment with your classmates at a high level. Any issues getting MySQL running is totally fine. What is not allowed is direct examination of anyone else's SQL code (on a computer, email, whiteboard, etc.) or allowing anyone else to see your SQL code.

You may use the search engine of your choice to look up the syntax for SQL commands or MySQL imperative SQL syntax, but may not use it to find answers.

You MAY post and discuss results with your classmates.

Ground Rules

1. Use the functions / trigger / stored procedure names specified.
2. Load data as provided

Loading the data

The file name `A5tableDefns.sql` contains all of the table creation statements. Run this file to create the tables.

The data are in `A5data.sql`, `sale.sql`, and `saleDetail.sql`

1 Functions

1. (10 points) Write a function, `numProductsAtEvent`, that returns the total number of products sold during an event. `NumProductsAtEvent` should take as an argument an eventId. If there is no event with the specified eventId, return -1.

   Execute the following statement and include the result in the comments in your homework submission.

   ```
   SELECT numProductsAtEvent(3);
   ```

2. (15 points) Write a function, `fractionXtra`, that computes the number of 'extra' toppings or flavors that were sold per eligible base product at each event. The table `xtra` has been provided to help you determine which 'extra' (or 'add-on') productCodes go with which base products. For example, all of the 'sundae' base products can have an 'sx' extra topping product. So, you want to tally how many products were sold

that could have an extra topping or flavor and determine what fraction of the those products actually DID have an extra topping or product sold.

`FractionXtra` should take an eventId as an argument and return a decimal, with 3 decimal places, indicating the fraction of extra items sold. If the specified event does not exist, return -1.

For example, if we have the following sales:

eventId	saleId	productCode
101	1334	c1
101	1335	cx
101	1336	d1
101	1337	dx
101	1339	wc

`SELECT fractionXtra(101);` would return 0.667.

Execute the following statement and include the result in the comments in your homework submission.

```
SELECT fractionXtra(4);
```

2 Triggers

1. (10 points) Recall, that some of our maintenance items need to occur after certain 'triggers.' For example, the ice cream machines must be cleaned after each event and the generator should be refueled after 40 hours of use.

   Table `eventLog` is used to record the actual start date time and the duration of each event, in minutes. This table is needed since some events may be delayed, cancelled or ended early due to a variety of reasons (e.g. rain, traffic, poor sales, etc.).

   Since we can't print messages in triggers in MySQL, we will create entries in the ToDo table when we need to perform some action. Our ice cream truck owner will regularly poll this table. The primary key for this table is automatically assigned. Set the value of the id attribute to the maintId value of the corresponding maintenance tasks. Set the toDoType value to 'maint'. Put the current date & time in recorded and

set done to FALSE. Once a task has been completed, the done attribute value will be set to TRUE by the person who completes the task.

Create a trigger named `afterEvent` that fires after inserts to the `eventLog` table. This trigger should populate the table `ToDo` when an "after event" maintenance rule needs to be executed.

Run the following statements and report your results:

```
INSERT INTO eventLog(eventId, actualStart, minutesDuration)
      VALUES (1,'2017-01-19 13:05:00' , 67);
SELECT * FROM toDo;
```

2. (15 points) Create a trigger, named `afterDelivery`, on that runs after updates to the delivery table. In this trigger, check to see if the deliveryDate field has been updated. If so, check to see if it is strictly more than 14 days after the order date. If it is, insert an entry in the ToDo table that indicates a late delivery. Use the deliveryId for the id attribute and set the toDoType to be 'late delivery'. Put the current date & time in recorded and set done to FALSE.

   Run the following statements and report your results:

```
TRUNCATE TABLE toDo;
UPDATE delivery
SET deliveryDate = '2018-01-16'
WHERE deliveryID = 2;

SELECT * FROM toDo;

TRUNCATE TABLE toDo;
UPDATE delivery
SET deliveryDate = '2018-02-16'
WHERE deliveryID = 2;

SELECT * FROM toDo;
```

3 Stored Procedures

1. (50 points) We want to know how well the power of suggestion works. If one person in line orders a particular product, are the next people

in line likely to order that same type of product or not? To determine this, we want to look at sequential product purchases.

Since product sales are entered into the sale table in the order in which they were purchased, we have an ordering for the products at each event.

Rather than comparing all possible combinations of the 27 products, we group similar products into product types. The table `ProductType` has been provided that maps each productCode to one of 9 product types.

Your stored procedure should populate the table `productTypePairs` with the counts of products, by type, that follow each other.

For example: If we sell a milkshake, then a pint of ice cream, then the value of 'ice cream beverage'–'pint' should be incremented by one. If we sell a pint of ice cream, and then a milkshake, the value of 'pint'–'ice cream beverage' should be incremented by one. Note that since we care about order, these two pairs are not equivalent. Note that your table should contain 'typeX'–'typeX' pairs as well. For example 'cone'-'cone'.

'Extra' products are special, since they are add-ons. To handle these, we skip any 'extra' products in our regular counts and then we handle them all by themselves. We only want to know about 'extra'–'extra' orderings. In this case we don't care what the base product was - we just want to know that if one person in line orders an extra 'something,' did the next person order something 'extra' as well? So, we want to count extra–extra pairs. At the product code level, 'slx'– 'wx' would increment our count, as would 'cx' – 'dx'. Create a single record in `productTypePairs` with productType1 = 'extra' and productType2 = 'extra' to store this count. Increment this count when you see an 'extra'–'extra' sequence or when there is at most one non-extra product between the two extra purchases.

For example, if we have the following sales:

eventId	saleId	productCode
42	1334	c1
42	1335	cx
42	1336	d1
42	1337	dx
42	1338	wx
42	1339	wc

This would count two 'extra'–'extra' pairs: one for cx and dx and one for dx and wx.

Chains should not cross event boundaries. In other words, the last product purchased at one event should not be paired with the first product purchased at the next event.

Write procedure `productChains` that implements the logic described above.