## Assignment 5

# A5: Imperative SQL

The goal of this assignment is to write functions 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 Postgres 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 PostrgreSQL 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 PostgreSQL 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 function and trigger 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.

Note that in these definitions we have added an attribute to sale: partOfSale. This attribute is usually NULL, but will be populated if someone purchases an add on product. For example, if someone buys a kid cone with an extra topping, the record for the extra topping will include the saleId of that kid cone.

The data are in `A5data.sql`, `sale20181104.sql`, and `saleDetail20181104.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. For example, cups of ice cream can have extra toppings. 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.

In this function 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 |

c1, d1, and wc all could have extra toppings. 2/3 of them do.

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

Execute the following statements and include the results in the comments in your homework submission.

```
SELECT fractionXtra(4);

SELECT fractionXtra(32);
```

3. (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 function 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 the function `productChains` that implements the logic described above.

## 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.

   Create a trigger named `printMaintAlert` that fires after inserts to the `truckEvent` table. This trigger should print the following message:

   ```
   New event  on <new date time>. Don't forget to get clean water!
   ```

   Where <new date time> should be the eventStart timestamp for the new event. The event should be allowed to be inserted.

   Run the following code to test your trigger. Include the results in a comment in your submission.

   ```
   INSERT INTO truckEvent(eventName, eventStart, plannedEnd)
   VALUES
   ('COMP430/530 party', '2018-11-30 2:00 PM', '2018-11-30 2:50 PM');
   ```

   In order to insert this record into the truckEvent table, you are likely going to need to reset the sequence object that assigns new SERIAL ids to the eventId attribute in truckEvent. You can use the `ALTER SEQUENCE` command to set the new value of the attribute to 1 past the maximum id value in the truckEvent table. You can look up the highest eventId value manually and execute this statement. It will only need to be run once. After that, the eventId should handle new inserts without conflicts.

2. (15 points) Event Booking Trigger We don't want to accidentally over book events.

   Add an attribute named `validEvent` to the truckEvent table, with a default value of 0. If this field is set to 1, the event is valid. If it is 0, the event is not valid and the owner has to decide what to do.

   Set the value of this field to 1 for all existing records in the table. Provide the SQL statement to do this!

   Next, add an insert trigger(s) to the truckEvent table that will insert the new record, but set the valid value to 0 if the new truckEvent overlaps with any existing event and will print the following error message:

```
New event  <eventName> on <new date time> overlaps with the following scheduled events:
```

and then lists the conflicts in order by eventStart.

If the new event does NOT overlap with any other event, set the valid flag to 1.

Finally, run the following query and include the results in your write-up:

```
INSERT INTO truckEvent(eventName, eventStart, plannedEnd)
VALUES
('Event1', '2017-06-16 14:30', '2017-06-16 20:45'),
('Event2', '2017-09-07 15:30', '2017-09-07 18:00'),
('Event3', '2017-09-07 17:30', '2017-09-07 21:00');


SELECT *
FROM truckEvent
WHERE eventName in( 'Event1', 'Event2', 'Event3')
ORDER BY eventName;
```