University of California, Los Angeles
Department of Computer Science

Computer Science 143 **Prof. Ryan Rosario**

Homework 3
Solutions.

Part 1: Joins and Subqueries

For these exercises we will write some queries to determine how much to charge Bird Scooter users for each ride they take. The tarball for HW3 contains some simulated (fake) data that you can use to test the syntax of your queries. You may notice that the the data types for these simulated relations are most `varchar` and there are no primary keys specified. This is so we do not give away the schema for those still completing HW2. We have the following relations:

```
customer(user_id, ccnum, expdate, email)
trip_start(trip_id, user_id, scooter_id, time, lat, lon)
trip_end(trip_id, user_id, scooter_id, time, lat, lon)
```

(a) Write a query that computes two new columns: (1) the elapsed time (in minutes) of each trip, and (2) the total cost of the trip. The trip charge is computed as follows: $1 flat rate for each trip plus $0.15 per minute. Fractional minutes should be rounded up (ceiling), so 4.02 minutes becomes 5 minutes. If the trip does not have an end time (scooter was stolen etc.), the length of the trip shall be 24 hours (1440 minutes) and the user should be charged based on 24 hours of use. Your results should include the `trip_id`, `user_id`, `trip_length` and `trip_cost`. There are at least two ways to do this problem. We prefer the method with the subquery and/or join. Compute the length of each trip first, and then compute the cost. Order the results by `trip_id` in ascending order. **Also, submit the top 10 rows of your output, without any special ordering.**

```
SELECT
  user_id,
  trip_id,
  trip_length,
  (1 + 0.15 * trip_length)::decimal(6,2) as trip_charge
FROM (
  SELECT
    L.user_id,
    L.trip_id,
    COALESCE(CEILING(EXTRACT(EPOCH FROM (R.time - L.time)) / 60), 1440) AS trip_length
  FROM trip_start L
  LEFT JOIN trip_end R
  ON L.trip_id = R.trip_id
) foo
ORDER BY trip_id
LIMIT 10;
```

Right join is also OK as long as the results match.

You should get the following output. Pay attention to the trip_cost column.

user_id	trip_id	trip_length	trip_cost
20685	0	2	1.30
34808	2	3	1.45
25463	3	1440	217.00
26965	4	2	1.30
836	5	1	1.15
3260	6	5	1.75
13850	7	3	1.45
23528	9	4	1.60
35829	11	1440	217.00
18494	12	1440	217.00

(b) Modify your query so that it computes the *total* amount that each user has spent on Bird Scooter. Again, use your previous response and assume that we did not store the intermediate result from the previous part. Report the user_id and the total amount spent as total_spent. Sort by user_id in ascending order. Be careful here.

```
SELECT
  user_id,
  SUM(trip_charge) AS total_spent
FROM (
  SELECT
    user_id,
    trip_length,
    SUM(1 + 0.15 * trip_length)::decimal(6,2) as trip_charge
    -- the method without the subquery requires repeating the COALESCE line in its entirety.
  FROM (
    SELECT
      L.user_id,
      L.trip_id,
      COALESCE(CEILING(EXTRACT(EPOCH FROM (R.time - L.time)) / 60), 1440) AS trip_length
    FROM trip_start L
    LEFT JOIN trip_end R
    ON L.trip_id = R.trip_id
  ) foo
  GROUP BY user_id
  ORDER BY trip_id
) bar
GROUP BY user_id
ORDER BY user_id DESC;
```

Right join is also OK as long as the results match. There should be at most one subquery.

You should get the following output:

user_id	total_spent
0	662.00
1	8.25
2	674.90
3	14.80
4	885.10
5	445.30
6	17.70
7	11.15
8	233.40
9	666.65

(c) In class, we learned several ways to classify joins. List all of the adjectives we can use to describe the join from part (a). Your choices are:

Inner	Outer	
Left	Right	Full
Equi	Non-equi	
Natural	Self	

Outer, left (or right, must be consistent with query), equi join.

(d) (Please submit this as a separate file `hw3.py`): Write a Python program that connects to the database, executes the query you wrote in the previous part, and produces a report that looks like the following. The format does not need to be identical, just close enough to show that you can use the output of the database and do something with it.

```
BIRD SCOOTER
User Charges for 2021

User ID        Charge
-----------    -----------
0               $ 662.00
1               $ 8.25
etc...
```

Hint: Practically all of the code is already written for you in the Lecture 8 slides. All you need to do is substitute in your query, eliminate the query parameter from my example, and then write some code to make the final computation.

Part 2: Views and Authorization

These are examples of short answer/short essay questions that may appear on the exam.

Exercises.

(a) In lecture, we discussed that RDBMS often grant authorizations based on users or *roles*, which are groups, and a user may be a member of zero or more groups. Suppose we have a role called `manager` and user `alice` has this role. First, how would `alice` and `manager` be represented in the authorization graph? How would the privileges `INSERT` and `SELECT` be represented?

Users and roles would be represented as nodes in the graph.

Different privileges $p_1, p_2, \ldots$ can either be represented in different graphs, or in the same graph with specific edge types representing each pirvilege grant. Either answer is fine.

(b) Alice can also grant privileges to other users of a database (`WITH GRANT OPTION`), and so can `manager`. But why would it be better for the granting to be done by the `manager` role rather than the `alice` user? Think in terms of the authorization model.

It's always better for a **role** to grant and revoke privileges, not a user. Alice could leave the company and then we have a problem. Cascading the `REVOKE` would not be proper because then others that Alice granted the privilege to that are still at the company lose the privilege. The subordinates will not lose the privilege if `manager` were the one to grant them. Upon exit, we just remove `alice` from the `manager` role.

(c) Explain some conditions when a standard `VIEW` cannot be made updatable. Why do you think that is?

From lecture, the query that creates the view cannot have `JOIN`, `GROUP BY`, or aggregation functions.

More specifically from the documentation:

```
The defining query of the view must have exactly one entry in the FROM clause,
which can be a table or another updatable view.
The defining query must not contain one of the following clauses at the top level:
GROUP BY, HAVING, LIMIT, OFFSET, DISTINCT, WITH, UNION, INTERSECT, and EXCEPT.
The selection list must not contain any window function , any set-returning function, or any
aggregate function such as SUM, COUNT, AVG, MIN, and MAX.
```

Remember that an updatable view updates the underlying table. If the query that creates the view involves a `JOIN`, we would need to insert data into two different tables. That's no so bad, except for the fact that by definition, a view applies to a single table.

What is worse is aggregation functions and `GROUP BY`. The view would contain aggregated data. The user would then be inserting **aggregate** data into an underlying table of **raw** data. If the user inserts a value for a column that is defined as an average, we have no way of mapping that value to the underlying raw data.