Homework 3 | Advanced SQL Queries

If updates are made to the assignment spec after its release, they are highlighted in red.

Objectives: To practice advanced SQL, using complex aggregates and/or subquerying, and gain experience with simple decorrelation.

Due date: Wednesday, Monday Oct. 22 @ 11:59pm

Resources

- Flights Schema document
- A SQL style guide in case you are interested (FYI only)

Part 1: SQL Queries

Below are some notes to keep in mind for this part:

- For each question, **write a single SQL query to answer that question**.
- **You should only use the parts of SQL we have covered in class**
    - With the exception that you may use any operators/functions from the documentation of SQLite)
    - ***NEW***: You CAN use subqueries/WITH (and are encouraged to!)
- Return the output columns exactly as indicated. Do not change the output column names, the output order, or return more/fewer columns.
- **Do not assume that attributes that aren't primary keys will be unique.**
- Unless otherwise noted, include canceled flights in your output.
- If a query uses a GROUP BY clause, ensure that all attributes in your SELECT clause are either grouping keys or aggregate values.
    - Recall that SQLite will let you select non-grouping-and-non-aggregate attributes. However, this is not standard SQL and other DBMSs would reject that query
- Although the provided dataset consists entirely of flights made in September 2024, *your queries should NOT assume this*. **We WILL test your queries on other datasets.**

Please review the above instructions before starting the problem and also before submitting your responses. The below problems are NOT in order of difficulty.

▶️ ***Submission Items*:** Put each query in a separate numbered .sql file, corresponding to the question number (`hw3-q1.sql`, `hw3-q2.sql`, etc).

1. *(Output relation cardinality: **22 rows**)*
   Find the 10 longest distinct distances flown. Then, for each such distance, find all offered "carrier routes", i.e., distinct (carrier, origin, destination) triples, and report how many times that carrier route was flown. Identify routes by airport codes, not city names. Sort the output in order of distance (highest distance first), breaking ties by origin (A-to-Z), then destination (A-to-Z), and then carrier name (A-to-Z).

   Name the output columns `carrier_name`, `origin`, `dest`, `distance_mi`, and `num_flights`.

2. *(Output relation cardinality: **1743 rows**)*
   For each carrier, calculate the percentage of its non-canceled flights that arrived on time or early (i.e., arrival delay <= 0). Sort the output in descending order of this percentage, and break ties in ascending alphabetical order by airline name. Report percentages as percentages, not decimals (e.g., report 75.2534... rather than 0.7525...). Do not round the percentages.

   Name the output columns `carrier_name` and `on_time_pct`. In case a carrier has no flights, include a row for that carrier with percentage 0.0.

   Hint: You can use CASE expressions to implement conditionals in SQL (kind of like if-statements or the ternary operator in other languages). You can put a CASE expression as the argument to an aggregate function.

3. *(Output relation cardinality: **70 rows**)*
   A "route" is an ordered pair of (origin airport code, destination airport code). (This is different from a "carrier route" which includes the carrier.)

   Find the 20 distinct most flown routes as well as every route that was flown exactly once. Show these results combined in one table, with columns `origin`, `origin_city`, `dest`, `dest_city`, and `num_flights`. Sort the output by `num_flights` from lowest to highest, and break ties by `origin` (A-to-Z), then `dest` (A-to-Z).

   Hint: to combine two relations with the same schema, check out the UNION keyword. Note carefully the limitations of UNION with regards to its component queries and LIMIT and ORDER BY. You may need to work around these limitations using additional subqueries or using the WITH keyword.

4. *(Output relation cardinality: **7 rows**)*The "competition factor" on a route is the number of distinct carriers who fly that route. For each distinct competition factor, find how many

distinct routes have that competition factor. Sort the output by competition factor from highest to lowest.

Name the output columns `competition_factor` and `num_routes`.

5. *(Output relation cardinality: **6 rows**)*
We consider a flight to be "timely" if it has an arrival delay of at most 15 minutes. The daily timeliness percentage for a carrier is the percentage of its flights for that day that are “timely”. Find the names of all carriers that flew at least one flight and have a daily timeliness percentage of at least 70% on every day they flew a flight. Sort in alphabetical order, and do not include duplicates. Cancelled flights can be included in the set of flights for that day, but are considered NOT “timely”.

   Name the output column `carrier_name`

6. *(Output relation cardinality: **252 rows**)*
Starting at some origin airport, a "direct destination" is an airport reachable in a single flight from the origin airport. The "connection factor" of an origin airport is how many distinct direct destinations one can reach from that origin.

   Find airport codes where all of their direct destinations have a strictly higher connection factor. Sort in alphabetical order, and do not include duplicates.

   Name the output column `airport`

7. *(Output relation cardinality: **367 rows**)*
For each origin airport, find the flight(s) with the longest duration (in minutes). If multiple flights from the same origin airport share the longest duration, include them all. Name the output column `origin`, `fid`, and `duration_mins`, in that order. We provide a sample solution using subqueries:

```
SELECT F1.origin, F1.fid, F1.duration_mins
FROM Flights F1
WHERE duration_mins = (
  SELECT MAX(duration_mins)
  FROM Flights AS F2
  WHERE F2.origin = F1.origin
);
```

   However, notice that this query is correlated, and thus takes an extremely long time to run in SQLite! You can verify this for yourself. Rewrite the solution into a more efficient **decorrelated** query. Your solution should not take more than 3 minutes to run on attu. For reference, our answer takes about 1 second.

Part 2: Reflection

This homework is relatively new, so the remaining questions help us to refine our specs for the future.

- How many hours did it take you to finish this assignment?
- How many of those hours did you feel were valuable and/or contributed to your learning?
- Did you collaborate with others on this assignment? If so, please write how many students you collaborated with, and their UW NetIDs.
- Is there anything you liked/disliked about this assignment? If not, please write N/A.

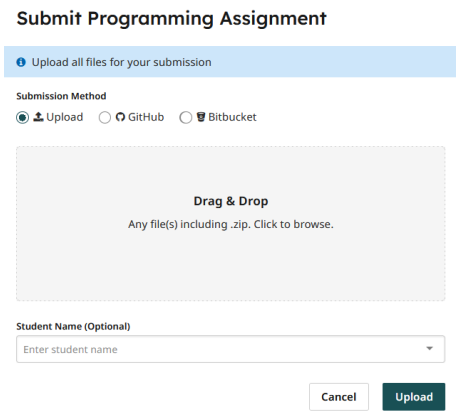
📂 ***Submission Item***: Please save your answers in a file called `hw3-reflection.txt`

Submission Instructions

What to turn in: `hw3-reflection.txt`, and `hw3-q1.sql`, ..., `hw3-q7.sql`

Please ensure that

- If you have any "dot commands" in your **query files** (q1-q7), such as `.headers on` or `.mode columns`, please remove these prior to submission
- You are submitting the .sql files directly to Gradescope, dragging the files to this screen (no zip file needed):



- Your file names match the expected file names (see above). Double check you have named all your files correctly!
    - **Do not submit your actual database (your .db file)**