Problem Set 2

Release Date: September 27, 2023

Due Date: October 6, 2023 by 11:59 pm ET

Submit to Gradescope: https://www.gradescope.com/courses/583077/assignments/3427030

You may work in pairs on this problem set. Only one of you needs to submit on Gradescope, but the other member must be added as a group member on the submission.

The purpose of this problem set is to give you some practice with concepts related to schema design, query planning, and query processing. Start early as this assignment is long.

Part 1 – Query Plans and Cost Models

In this part of the problem set, you will examine query plans that PostgreSQL uses to execute queries, and try to understand why it produces the plan it does for a certain query.

We are using the same dataset as in problem set 1. We have loaded the MBTA dataset into a Postgres server hosted on AWS that you'll be using for this problem set. To access the server, you can start a session with:

```
psql -h mbta-db-class.c39astlavjy2.us-east-1.rds.amazonaws.com -p 5432 -d mbta -U beaver
```

When prompted for a password, use "student" (without the quotes).

To use PostgreSQL, you can `ssh` into `athena.dialup.mit.edu` and get started. In case you want to work on your own machine, you can install the PostgreSQL client.

If you are having trouble connecting to the database and you are not using Athena, try connecting when you are on the MIT network (i.e., be on MIT SECURE or be connected to the VPN).

To help understand database query plans, PostgreSQL includes the `EXPLAIN` command. It prints the physical query plan for the input query, including all of the physical operators and internal access methods being used. For example, the SQL command displays the query plan for a very simple query:

```
EXPLAIN SELECT * from routes;
```

```
                                 QUERY PLAN
-----------------------------------------------------------------------------
    Seq Scan on routes  (cost=0.00..1.18 rows=18 width=67)
(1 row)
```

To be able to interpret plans like the one above, you can refer to the EXPLAIN section in the Postgres documentation.

We have run `VACUUM FULL ANALYZE` on all of the tables in the database, which means that all of the statistics used by PostgreSQL server should be up to date.

To identify an index, it is sufficient for you to name the ordered sequence of columns that are indexed. E.g., an index on columns *foo* and *bar* is identified as *(foo, bar)*.

Include the output of `EXPLAIN` or `EXPLAIN ANALYZE` queries if relevant. Be as brief as possible.

1. [4 points]: What are the columns of table `lines`? What indexes exist for the table `routes` in `mbta`? What is the primary key (or primary key pair) of table `gated_station_entries`? You can use the `\?` and `\h` commands to get help, and `\d <tablename>` to see the schema for a particular table.

2. [4 points]: What query plan does Postgres choose for

```
SELECT route_id FROM routes;
```

Is it different from the plan for

```
SELECT * FROM routes;
```

shown above? Given the indexes we have defined on the table, are there any other possible query plans?

3. [4 points]: What query plan does Postgres choose for

```
SELECT station_id FROM gated_station_entries;
```

Is it different from the plan for

```
SELECT station_id FROM gated_station_entries ORDER BY station_id;
```

If so, why are they different? If not, why are they the same?

We now create a few new versions of the table `station_orders` to study the accuracy of cardinality (number of output rows) estimation.

1. `station_orders1`: This is an exact copy of `station_orders`. We created it by executing (you cannot execute this because you don't have write access):

```
CREATE TABLE station_orders1 AS (SELECT * FROM station_orders);
```

2. `station_orders2`: We removed the tuples with route_id > 12 from station_orders. We set autovacuum_enabled = false to ensure Postgres's statistics do not reflect the fact that we removed tuples where route_id > 12 (you can read about "vaccuuming" at `https://www.postgresql.org/docs/current/routine-vacuuming.html`). We created this table using the following commands (again, you cannot execute these):

```
CREATE TABLE station_orders2 AS (SELECT * FROM station_orders);
ALTER TABLE station_orders2 SET (autovacuum_enabled = false);
DELETE FROM station_orders2 WHERE route_id > 12;
```

4. [6 points]: Analyze the query plans for the following queries on each of the tables described above.

```
EXPLAIN SELECT * from station_orders1 WHERE route_id > 10;
EXPLAIN SELECT * from station_orders2 WHERE route_id > 10;
```

Is there a difference between the estimated rows for the queries? Which is one more accurate? Why is that? [HINT: use SELECT COUNT(*) ... to find the exact number output rows]

Notice that the first query estimates the exact number of output rows without actually executing the select *. Why do you think it is the case? (Note that we will reveal the exact reason in cardinality estimation lecture. For now, we give full credit on any reason you think is reasonable.)

Notice that the query plan in each case is the same. Does this mean that the out-of-date statistics for `station_orders2` do not matter? Can you describe at a high-level another query where the out-of-date statistics for `station_orders2` may lead to a bad query plan?

Consider the following two queries:

```
SELECT COUNT(time)
FROM gated_station_entries
WHERE station_id = 'place-haecl'
AND line_id = 'green';
```

and

```
SELECT COUNT(time)
FROM gated_station_entries
WHERE station_id ILIKE '%place-haecl%'
AND line_id = 'green';
```

5. [6 points]: Both queries produce the same results, however, they have very different performance characteristics. Use `EXPLAIN` and `EXPLAIN ANALYZE` to look at the query plans, and the performance (execution time) of these two queries. Provide the `EXPLAIN` output for both the query plans, and describe what they do. Why is the first query faster to execute?

Now consider the queries generated by replacing `TOTAL` in the below query with '10000' and '400000' in the following template. (You can refer to the two queries as Q10k and Q400k, respectively.)

```
EXPLAIN SELECT COUNT(*)
        FROM rail_ridership
        JOIN time_periods using(time_period_id)
        JOIN lines using(line_id)
        WHERE lines.line_name = 'Green Line'
        AND time_periods.day_type = 'weekday'
        AND rail_ridership.total_ons >= TOTAL;
```

6. [4 points]: What physical plan does PostgreSQL use for each of them? Your answer should consist of a drawing of the two query trees and annotations on each node. [Note that: "materialize" operator stores the result of its child in memory. You can ignore this operator when drawing your plan trees.]

7. [2 points]: What access methods are used? (also label them in the diagrams)

8. [2 points]: What join algorithms are used? (also label them in the diagrams)

9. [2 points]: By running `EXPLAIN ANALYZE` on Q10k, can you see if there are any final or intermediate results where PostgreSQL's estimate is less than half or more than double the actual size?

10. [10 points]: Vary the values of `TOTAL` in increments of 40000 from 10000 to 400000 and briefly describe how the plans change. (You do not need to show all of the plans or list out the behavior for all of them. Just describe when Postgres switches plans.) Why do you think the plan changes? [HINT: there are 4 different plans in total.]

Part 2 – Query Plans and Access Methods

In this problem, your goal is to estimate the cost of different query plans and think about the best physical query plan for a SQL expression.

TPC-H is a common benchmark used to evaluate the performance of SQL queries. It represents orders placed in a retail or online store. Each `order` row relates to one or more `lineitem` rows, each of which represents an individual `part` record purchased in the order. Each `order` also relates to a `customer` record, and each `part` is related to a particular `supplier` record. Each `customer` belongs to a `nation`.

A diagram of the schema of TPC-H is shown in Figure 1 (note that the table sizes are given in this diagram).

In addition to specifying these tables, the benchmark describes how data is generated for this schema, as well as a suite of about 20 queries that are used to evaluate database performance by running the queries one after another.

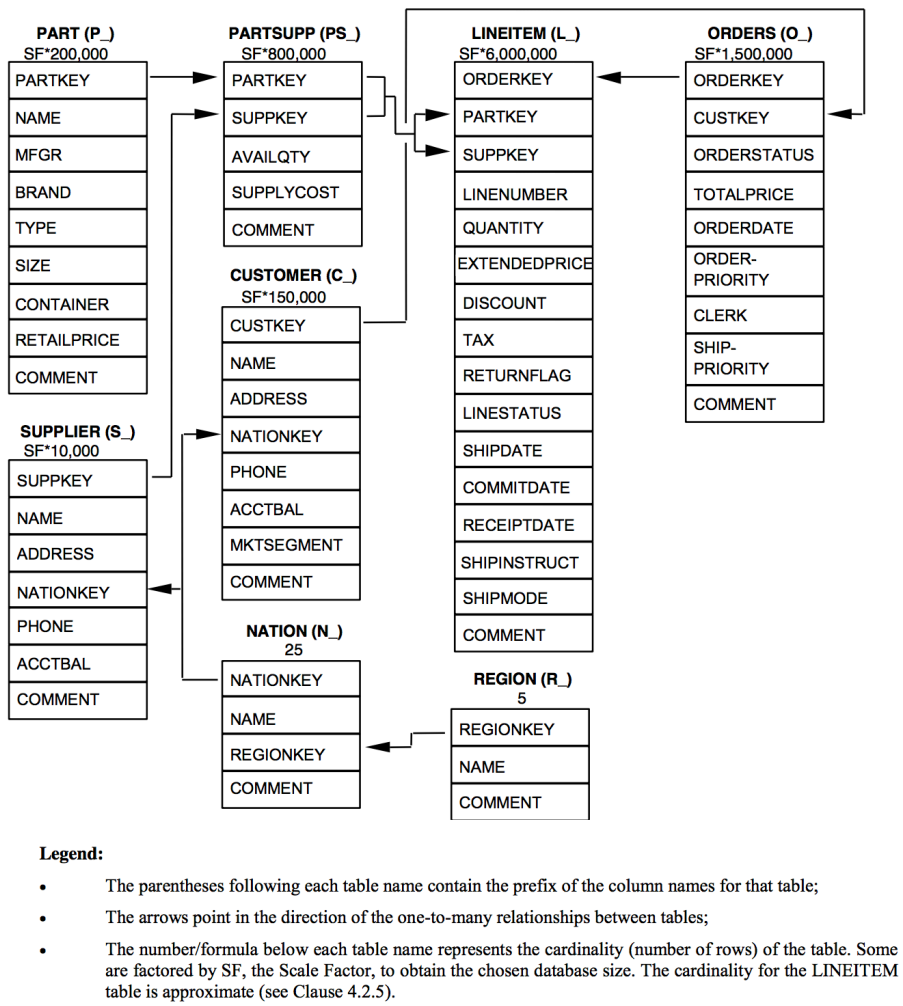

Figure 1: The TPC-H Schema (source 'The TPC-H Benchmark. Revision 2.17.1')

TPC-H is parameterized by a "Scale Factor" or "SF", which dictates the number of records in the different tables. For example,

for SF=100, the `lineitem` table will have 15 million records, since the figure shows that `lineitem` has size SF*150,000.

Consider the following query, which is a modified version of Query 5 in the TPC-H benchmark. For each country in a given region, this query retrieves the (i) total revenue that arose from transactions that had both the customer and the supplier in that nation, and (ii) a comment that describes the nation in question.

```sql
SELECT
    n_name,
    n_comment,
    count(*)
FROM
    customer,
    orders,
    lineitem,
    supplier,
    nation
WHERE
    c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND l_suppkey = s_suppkey
    AND c_nationkey = s_nationkey
    AND s_nationkey = n_nationkey
    AND n_name in ('GERMANY', 'UNITED STATES')
    AND l_shipmode = 'AIR'
    AND o_orderdate >= date '1994-01-01'
    AND o_orderdate < date '1994-07-01'
    AND o_orderpriority = '1-URGENT'
    AND c_mktsegment = 'MEDIUM PLATED STEEL'
GROUP BY
    n_name
ORDER BY
    revenue DESC;
```

Your job is to evaluate the best query plan for this query. To help you do this, we provide some basic statistics about the database:

- A `customer` record is 179 bytes, an `orders` record is 104 bytes, a `lineitem` record is 112 bytes, a `supplier` record is 159 bytes, a `nation` record is 128 bytes, and a `region` record is 124 bytes. (As outlined in Section 4.2.5 of the TPC-H spec.) Thus, an SF=100 `customer` table takes 179 * 15M = 2.685 GB of storage. (Note that the number of `region` and `nation` values is fixed. As seen in the SF multiplicative factor in the figure above, those tables do not grow with the scale factor.)
- All key attributes are 4 bytes, all numbers are 4 bytes, dates are 4 bytes, the `n_name` field is a 25 byte character string (assume strings are fixed length) the `n_comment` field is a 117 byte character string, the `l_shipmode` and `c_mktsegment` fields are a 10 byte character string, and `o_orderpriority` is a 14 byte character string.
- `c_custkey` is the primary on table `customer`, `o_orderkey` is the primary on table `order`, `n_nationkey` is the primary on table `nation`, `s_suppkey` is the primary on table `supplier`.
- `n_name` is one of 25 distinct string values chosen uniformly at random.
- `o_orderdate` values in the database are selected uniformly random between '1992-01-01' and '1998-12-31' - 151 days.
- `l_shipmode` is one of seven distinct string values chosen uniformly at random.
- `o_orderpriority` is one of five distinct string values chosen uniformly at random (1-URGENT, 2-HIGH, 3-MEDIUM, 4-NOT SPECIFIED, or 5-LOW)
- `c_mktsegment` is string uniformly random selected out of a list of 150 strings.

You create these tables at scale factor SF=100 in a row-oriented database. The system supports heap files and B+-trees (clustered and unclustered). B+-tree leaf pages point to records in the heap file. Assume you can cluster each heap file in according to

exactly one B+tree, and that the database system has up-to-date statistics on the cardinality of the tables, which can be used to accurately estimate the selectivity of each filter predicate.

Assume disk seeks take 1 ms, and the disk can sequentially read 1 GB/sec. In your calculations, you can assume that I/O time dominates CPU time (i.e., you do not need to account for CPU time.)

Your system has a 4 GB buffer pool, and an additional 10 GB of memory to use for buffers for joins and other intermediate data structures. Assume the page size is 64 KB and further assume that indexes and hash tables do not take any space in memory or buffer pool.

Finally, suppose the system has grace hash joins, indexed nested loop joins, and simple nested loop joins available to it.

11. [6 points]: Suppose you have no indexes. Draw (as a query plan tree) what you believe is the best query plan for the above query. For each node in your query plan indicate (on the drawing) the approximate output cardinality (number of tuples produced). For each join indicate the best physical implementation (i.e., grace hash or nested loops). You do not need to worry about operations such as grouping / aggregation / sort/ hash. [HINT: Keep in mind that your optimal plan should have the intermediate result size as small as possible.] [HINT: Recall in lecture 8 page 10 that for primary key-foregin key join, the selectivity of the join predicate is 1/ICARD(PK table). In this case, can just assume ICARD(PK table) = NCARD(PK table)].

12. [4 points]: Estimate the runtime of the query in seconds (considering just I/O time).

13. [6 points]: If you are only concerned with running this query efficiently, which indexes, if any, would you recommend creating? How would you cluster each heap file? Note that you can ignore the cost of creating an index or clustering heap files.

14. [4 points]: Draw (as a query plan tree), what you believe is the best query plan for the above query given the indexes and clustering you chose. For each node in your query plan indicate (on the drawing, if you wish), the approximate output cardinality (number of tuples produced.) For each join indicate the best physical implementation (i.e., grace hash, nested loops, or index nested loops).

15. [4 points]: Estimate the runtime of the query in seconds once you have created these indexes/clustered heap files (considering just I/O time).

Part 3 – Schema Design and Query Execution

Suppose you are building a relational database to record information about pets and their owners, their vets, and the shows the pets participate in. Specifically, you want to record:

- For each pet, its species, breed, fur color, and eye color.
- For each owner, their name and address.
- For each vet, their name and address.
- For each show, the title of the show, it's date, and location.

Further you want to capture the following relationship information:

- Pets have zero or more owners and owners have zero or more pets.
- Vets treat zero or more pets, and pets are treated by exactly one vet.
- Pets visit the vet zero or more times each year. Each visit has a date and a cost.
- Pets compete in zero or more shows, and each show has zero or more pet competitiors.
- Pets can win zero or one prizes in each show. A prize is just a string, e.g., "best in show".

16. [6 points]: Draw an entity relationship (ER) diagram that captures as many of the properties above as possible. You may need to make some assumptions about the nature of the relationships between different entities.

17. [6 points]: Write out a schema for your database based on your ER Diagram. Include a few sentences of justification for why you chose the tables you did.

18. [4 points]: Is your schema redundancy and anomaly free? Justify your answer.

19. [8 points]: Dana Bass insists that a hierarchical schema would be a better representation for this data. Is he right? Justify your answer in a few sentences.