Problem Set 2

Release Date: September 28, 2022

Due Date: October 17, 2022 by 11:59 pm ET

Late Due Date: October 22, 2022 by 11:59 pm ET

Submit to Gradescope: `https://www.gradescope.com/courses/421252`

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 that you'll be using for this problem set. To access the server, you can start a session with:

```
psql -d mbta -h dsg-burton.csail.mit.edu -p 8000 -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:

```sql
EXPLAIN SELECT * from gated_station_entries;
```

```
                                  QUERY PLAN
---------------------------------------------------------------------------
 Seq Scan on gated_station_entries  (cost=0.00..161284.03 rows=8483703 width=40)
(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. [2 points]: What indexes exist for the table `gated_station_entries` in `mbta`? 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

```sql
SELECT station_id FROM gated_station_entries;
```

Is it different from the plan for

```sql
SELECT * FROM gated_station_entries;
```

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

3. [2 points]: In one sentence, describe the difference between the plan from the previous question and the plan for:

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

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

```sql
SELECT station_id, gated_entries FROM gated_station_entries ORDER BY station_id;
```

Is it different from the plan for

```sql
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 `rail_ridership` that mimic the process of updating data. We will study how data updates and statistics interact.

1. `rail1`: This is an exact copy of `rail_ridership`, without statistics or indices. We can get it by executing:

   ```sql
   CREATE TABLE rail2 AS (SELECT * FROM rail_ridership);
   ```

2. `rail2`: This is the same as `rail1`, but after running `VACUUM ANALYZE`.
3. `rail3`: We create this table by deleting the rows from `rail2` that are on the green line.

   ```sql
   DELETE FROM rail2 WHERE rail2.line_id = 'green';
   ```

   Note that we have configured PostgreSQL to have `autovacuum = off`. Setting `autovacuum = on` would have automatically updated the statistics after deletion. However, this setting would also make deletions more expensive. This example shows a typical trade-off between having accurate statistics versus fast database updates. You can read more about what Postgres does during a vacuum by looking at the documentation (`https://www.postgresql.org/docs/current/sql-vacuum.html`).

5. [4 points]: Analyze the query plans for the following query on each of the 3 tables described above. In particular, describe how the row size estimate for the filter changes:

```
EXPLAIN SELECT * from rail1 AS r WHERE r.line_id = 'green';
EXPLAIN SELECT * from rail2 AS r WHERE r.line_id = 'green';
EXPLAIN SELECT * from rail3 AS r WHERE r.line_id = 'green';
```

Notice that the query plan in each case is the same. Does this mean that the out-of-date statistics for `rail1` or `rail3` do not matter? Can you give an example of another query where the out-of-date statistics for `rail1` or `rail3` lead to a bad query plan?

6. [4 points]: What query plan does Postgres choose for

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

Is Postgres's optimizer correct? If not, what do you think it is doing wrong, or perhaps what would need to change to fix it? How do the actual performance of other possible plans compare?

You can coerce Postgres into using a variety of query plans by using the `enable_seqscan`, `enable_indexonlyscan`, `enable_indexscan` and the `enable_bitmapscan` flags with the `set` command. For example,

```
set enable_indexonlyscan=off;
```

will prevent the query optimizer from choosing an index only scan. Furthermore, you can configure the internal cost model by setting parameters like `seq_page_cost` and `random_page_cost` (for the cost of sequential and random reads, respectively) to encourage the optimizer to choose plans which are better for the machine.

Consider the following two queries:

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

and

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

7. [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 of these two queries. Paste the `EXPLAIN` output for both the query plans, and describe what they do. Why is the first query much faster to execute?

Now consider the queries generated by replacing `TOTAL` in the below query with '60000' and '180000' in the following template. (You can refer to the two queries as Q60k and Q180k respectively.)

```
EXPLAIN SELECT COUNT(*)
FROM stations
JOIN station_orders
ON stations.station_id = station_orders.station_id
JOIN rail_ridership
ON rail_ridership.station_id = stations.station_id
WHERE rail_ridership.total_ons >= TOTAL;
```

8. [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.

9. [2 points]: What access methods are used? (also label them in the diagrams)

10. [2 points]: What join algorithms are used? (also label them in the diagrams)

11. [4 points]: By running some queries to compute the sizes of the intermediate results in the query, and/or using `EXPLAIN ANALYZE`, 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?

12. [10 points]: Vary the values of `TOTAL` in increments of 40000 from 60000 to 500000 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.) Do you believe the query planner is switching at the correct points?

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.

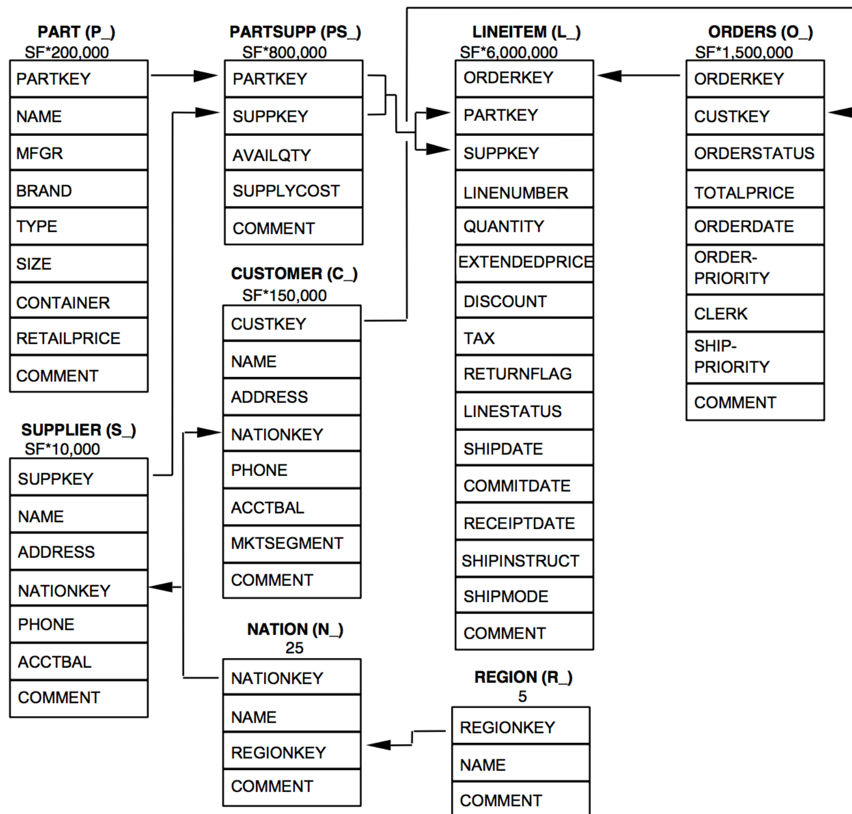

Figure 1: The TPC-H Schema (source 'The TPC-H Benchmark. Revision 2.17.1')

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.

```
SELECT
    n_name,
    n_comment,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue
FROM
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
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_regionkey = r_regionkey
    AND r_name = 'ASIA'
    AND l_shipmode = 'MAIL'
    AND o_orderdate >= date '1993-01-01'
    AND o_orderdate < date '1993-01-01' + interval '1' year
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, and the `l_shipmode` field is a 10 byte character string.
- `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.
- `l_discount` is uniformly random between 0.00 and 1.00.
- `l_extendedprice` is uniformly and randomly distributed between 90000 and 111000 (in reality the price computation in TPC-H is somewhat more complex, but this is approximately correct).

You create these tables at scale factor 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, and can accurately estimate the selectivity of every predicate. Assume B+-tree pages are 50% full, on average.

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.

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

13. [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, 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). You do not need to worry about the implementation of the grouping / aggregation operation.

14. [4 points]: Estimate the runtime of the query in seconds (considering just I/O time).

15. [6 points]: If you are only concerned with running this query efficiently, and insert time is not a concern, which indexes, if any, would you recommend creating? How would you cluster each heap file?

16. [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).

17. [4 points]: Estimate the runtime of the query in seconds once you have created these indexes (considering just I/O time).

Part 3 – Schema Design and Query Execution

Suppose you are building a relational database to record information about actors and movies. Specifically, you want to record:

- For each movie, its title, release year, and genre.
- For each actor, their first and last names, birth date, and gender (1 character).
- For each director, their first and last names and gender (1 character).

Further you want to capture the following relationship information:

- Movies have zero or more directors, and each director directs zero or more movies.
- Each movie has zero or more actors, and each actor acts in zero or more movies.
- For each actor in each movie, you also want to record the role of the actor.
- Movies may be the sequel of zero or one other movies.

18. [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.

19. [6 points]: Write out a schema for your database in BCNF. You may include views. Include a few sentences of justification for why you chose the tables you did.

20. [4 points]: Is your schema redundancy and anomaly free? Justify your answer.

21. [8 points]: Ben Bitdiddle insists that a hierarchical schema would be a better representation for this data. Is he right? Justify your answer in a few sentences.