Problem Set 2

Release Date: March 9, 2021

Due Date: 11:59 pm ET March 29, 2021

Late Due Date: 11:59 pm ET April 2, 2021

Submit to Gradescope: `https://www.gradescope.com/courses/229885`

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 problem set 1. We have loaded the Olympics 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 -h istc12.csail.mit.edu -p 5432 -U student olympics
```

Make sure your PostgreSQL client is 10.15+ so that your results are consistent with the solutions. Athena already has client version 10.15 installed, so you can `ssh` into `athena.dialup.mit.edu` and get started. In case you want to work on your own Debian/Ubuntu machine, you can install the `postgresql-client` package by running the following command in your shell.

```
sudo apt-get install postgresql-client
```

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 athletes;
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on athletes  (cost=0.00..2473.71 rows=135571 width=38)
(1 row)
```

To be able to interpret plans like the one above, you can refer to the EXPLAIN section in the Postgres documentation. **Note that we are using PostgreSQL 9.6**.

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. Eg., an index on columns *foo* and *bar* is identified as *(foo, bar)*.

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

1. [2 points]: What indexes exist for the table `athletes` in `olympics`? 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 name FROM athletes;`? Is it different from the plan for `SELECT * FROM athletes;` 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 `SELECT name FROM athletes ORDER BY name;`.

4. [4 points]: What query plan does Postgres choose for `SELECT name, age FROM athletes ORDER BY name;`? Is it different from the plan for `SELECT name FROM athletes ORDER BY name;`? If so, why are they different? If not, why are they the same?

We now add another table `athletes_wide` into `olympics`. The table `athletes_wide` contains the same rows as `athletes`, except that it has an additional column with type `CHAR(100000)`. If you want to know more about the contents of the two tables, you can examine them further using `\d <tablename>` and your own SQL queries. Now, consider the two following queries and their plans.

```
EXPLAIN SELECT name from athletes;
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on athletes  (cost=0.00..2473.35 rows=135535 width=20)
(1 row)

EXPLAIN SELECT name from athletes_wide;
                            QUERY PLAN
------------------------------------------------------------------
 Index Only Scan using athletes_wide_name on athletes_wide
        (cost=0.42..4685.76 rows=135556 width=19)
(1 row)
```

5. [4 points]: Both of the queries only need to use the value of a column which has already been indexed. From the perspective of PostgreSQL's optimizer, why does it choose a sequential scan on `athletes` but an index only scan on `athletes_wide`?

6. [4 points]: Now run this scan on both tables, using index only scans and sequential scans. How does their actual performance compare? Is Postgres's optimizer correct? If not, what do you think it is doing wrong? 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.

Consider the following two queries and their plans from `olympics`:

```
EXPLAIN ANALYZE SELECT team, noc FROM athlete_events
        WHERE noc < 'R' and noc > 'QB';
                        QUERY PLAN
-----------------------------------------------------------------
 Index Only Scan using athlete_events_team_noc on athlete_events
        (cost=0.42..7167.09 rows=131 width=13)
        (actual time=78.631..78.631 rows=0 loops=1)
   Index Cond: ((noc < 'R'::text) AND (noc > 'QB'::text))
   Heap Fetches: 0
 Planning time: 0.168 ms
 Execution time: 78.664 ms
(5 rows)

EXPLAIN ANALYZE SELECT team, noc FROM athlete_events
        WHERE team < 'R' and team > 'QB';
                        QUERY PLAN
-----------------------------------------------------------------
Index Only Scan using athlete_events_team_noc on athlete_events
        (cost=0.42..7.20 rows=139 width=13)
        (actual time=0.097..0.106 rows=21 loops=1)
   Index Cond: ((team < 'R'::text) AND (team > 'QB'::text))
   Heap Fetches: 0
 Planning time: 0.312 ms
 Execution time: 0.141 ms
(5 rows)
```

7. [6 points]: The two queries and their plans are very similar and make use of the same index. Why are the costs (both the estimates and actual) so different?

Now consider the queries generated by replacing `YEAR` in the below query with '1900' and '2012' in the following template. (You can refer to the two queries as Q1900 and Q2012 respectively.)

```
EXPLAIN SELECT AVG(age)
FROM athletes
JOIN athlete_events
  ON athletes.id = athlete_events.id
JOIN host_cities
  ON athlete_events.games = host_cities.games
WHERE age IS NOT NULL AND host_cities.year > YEAR;
```

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 `YEAR` in increments of 4 from 1896 to 2016 and briefly describe how the plans change. (you don't need to show all of the plans or list out the beahvior for all of the years - just describe when it switches plans.) Do you believe the query planner is switching at the correct points? (Justify your answer quantitatively).

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.

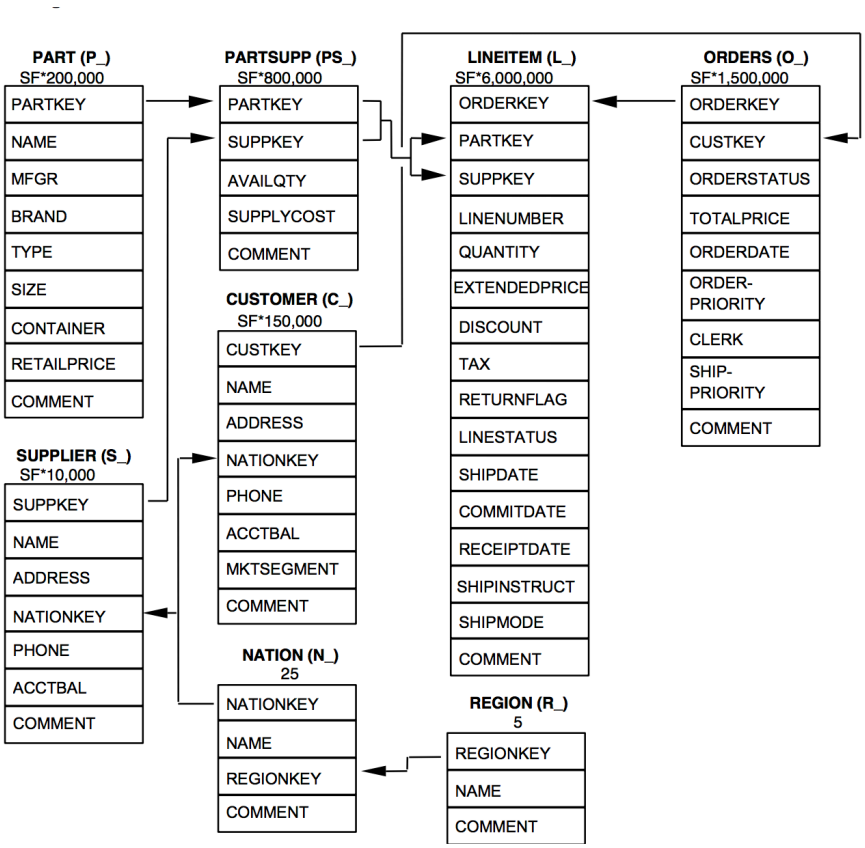

Figure 1: The TPC-H Schema (source 'The TPC-H Benchmark. Revision 2.17.1')

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 600 million records, since the figure shows that `lineitem` has size SF*6,000,000.

Consider the following query, representing Query 10 in the TPC-H benchmark, which computes the total (lost) revenue by customer from items that were returned in a 3 month interval.

```
SELECT
    c_custkey,
    c_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
FROM
    customer,
    orders,
    lineitem,
    nation
WHERE
    c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate >= date '1993-10-01'
    AND o_orderdate < date '1993-10-01' + interval '3' month
    AND l_returnflag = 'R'
    AND c_nationkey = n_nationkey
GROUP BY
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
ORDER BY
    revenue desc
LIMIT 20;
```

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 `lineitem` record is 112 bytes, an `order` record is 104 bytes, a `nation` record is 128 bytes, and a `customer` record is 179 bytes, as outlined in Section 4.2.5 of the TPC-H spec. Thus, an SF=100 `lineitem` table takes 112 * 600M = 67.2 GB of storage.
- All key attributes are 4 bytes, all numbers are 4 bytes, dates are 4 bytes, the `l_returnflag` field is a 1 byte character string, `c_address` field is a 40 byte character string,`c_phone` field is a 15 byte character string, `c_comment` field is a 117 byte character string, and the `n_name` field is a 25 byte character string (assume strings are fixed length).
- `o_orderdate` is selected uniformly random between '1992-01-01' and '1998-12-31' - 151 days.
- `l_returnflag` is either 'R' or 'A', selected uniformly and 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 10 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 or 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

The Blings are an international Crime Syndicate specializing in theft. The Blings focus their attacks only on jewelry stores all over the world. In recent time, Blings has grown into a massive organisation and it has become difficult for them to track the logistics of their crimes.

In order to solve this issue, the Blings spy force invades MIT and takes you prisoner. They threaten to cut off your bubble tea supply unless you help them design a relational database to keep track of the logistics for their attack.

Specifically, they want to keep track of:

- The address, size(in sq ft.), number of employees, and approximate value of the jewelry at each jewelry store.
- The names and ranks of their members who will rob each store. Many members can rob each store, but each store is robbed only once. Each member can only rob one store. Each store should be robbed by at least one member.
- Blings members carry the stolen jewelry and deposit it to the office. The jewelry is of one of several types. Each type has a color and a quality metric(on a 1–10 scale).
- Each piece of jewelry has a type and is carried by exactly one member. Each member may deposit multiple pieces of jewelry.
- The dates and times when they plan to rob each store, and the members that will participate in each robbery.

Blings member Root (who is not very bright) started the database design task, and has come up with this partial schema to represent information about Blings members and their jewelry.

```
{members (member_name | member_rank | jewelry | jewelry_quality )}
```

For example, some entries might include:

```
bess, general, necklace, 10
stokes, general, bracelet, 8
```

18. [4 points]: List three problems that might arise with Root's schema as the database grows and members are promoted, arrested, removed or acquire more jewelry.

19. [6 points]: You agree to help Blings, because you know that bubble tea is love. You tell them the first step in good schema design is an Entity Relationship (ER) diagram. Draw one that captures as many of the properties of their operations above as possible (not just the properties of the members and jewelry as in Root's database.) You may need to make some assumptions about the nature of the relationships between different entities.

20. [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.

21. [4 points]: Is your schema redundancy and anomaly free? Justify your answer.

22. [8 points]: Root, who was a member of the IBM IMS team in a past life, insists that a hierarchical schema would be a better representation for this data. Is Root right? Justify your answer in a few sentences.