6.830 Problem Set 2 (2017)

Assigned: Monday, Sep 25, 2017

Due: Monday, Oct 16, 2017 , 11:59 PM
Submit to Gradescope: `https://gradescope.com/courses/10498`

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 for Problem Set 1 (we loaded the PS1 SQLite data into the Postgres server that you'll be using for PS2). To access the server, you can log in to `athena.dialup.mit.edu` and start a session with:

```
psql -h geops.csail.mit.edu -p 5433 imdb
```

Make sure your PostgreSQL client is 9.3+ so that your results are consistent with the solutions. Athena already has client version 9.3.9 installed, so you can simply `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 movies;
                          QUERY PLAN
----------------------------------------------------------------
Seq Scan on movies  (cost=0.00..8418.65 rows=464065 width=36)
(1 row)
```

To be able to interpret plans like the one above, you should refer to the explain basics 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.

Note: To identify an index, it is enough 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)* .

1. [1 points]: Which indexes exist for table `movies` in `imdb`? You can use the `\?` and `\h` commands to get help, and `\t <tablename>` to see the schema for a particular table.

2. [2 points]: Which query plan does Postgres choose for `select title from movies`? Is it different from the plan shown in the previous page? Given the indexes we have defined on our table, are there any other possible query plans?

3. [1 points]: In one sentence, describe the difference between the plan from the previous question and the plan for query: `select title from movies order by title`.

4. [2 points]: What query plan does Postgres choose for `select title, year from movies order by title`? Is it different from the plan for `select title from movies order by title`? If so, why are they different? If not, why are they the same?

We now add another two tables, `people_reduced` and `people_wide`, into `imdb`. The table `people_reduced` simply contains 1000 rows from `people` (the reduced table size is not significant with regard to your answer, we just needed to decrease loading time for `people_wide` and wanted to keep the size constant between `people_reduced` and `people_wide` for comparison). The table `people_wide` contains the same 1000 rows as `people_reduced`, except that it has an additional column with type `character(20000)`. If you want to know more about the contents of the two tables, you can examine them further using `\d [table-name]` and your own SQL queries. Now, consider the two following queries and their plans from `imdb`:

```
explain select name from people_reduced;
                            QUERY PLAN
-------------------------------------------------------------------
Seq Scan on people_reduced  (cost=0.00..18.00 rows=1000 width=14)
(1 row)
```

```
explain select name from people_wide;
                       QUERY PLAN
---------------------------------------------------------------
Index Only Scan using people_wide_name on people_wide
    (cost=0.28..43.27 rows=1000 width=14)
(1 row)
```

5. [2 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 seq scan on `people_reduced` but an index only scan on `people_wide`?

6. [2 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.

7. [3 points]: Consider the two following queries and their plans from `imdb`:

```
explain analyze select movie_id, person_id
    from cast_members
    where movie_id='tt0120737';
                                QUERY PLAN
----------------------------------------------------------------------
Index Only Scan using cast_members_pkey on cast_members
(cost=0.43..4.59 rows=9 width=20)
(actual time=0.062..0.065 rows=10 loops=1)
  Index Cond: (movie_id = 'tt0120737'::text)
  Heap Fetches: 0
Planning time: 0.160 ms
Execution time: 0.088 ms
(5 rows)


explain analyze select movie_id, person_id
    from cast_members
    where person_id='nm0000704';
                                QUERY PLAN
----------------------------------------------------------------------
Index Only Scan using cast_members_pkey on cast_members
(cost=0.43..91110.02 rows=18 width=20)
(actual time=57.129..282.138 rows=45 loops=1)
   Index Cond: (person_id = 'nm0000704'::text)
   Heap Fetches: 0
Planning time: 0.261 ms
Execution time: 282.189 ms
(5 rows)
```

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 'a' in the below query with '2005' and '2050' in the following template. (You can refer to the two queries as Q2005 and Q2050 respectively. Also don't worry about how strange it is that imdb stores movies with years so far into the future, just treat the movies with future years the same as you would any other movie).

```
explain select avg(birth_year) from people
    join cast_members on people.id = cast_members.person_id
    join movies on cast_members.movie_id = movies.id
    where people.birth_year is not null and movies.year > a;
```

8. [2 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. [1 points]: Which access methods are used? (also label them in the diagrams)

10. [1 points]: Which join algorithms? (also label them in the diagrams)

11. [2 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?

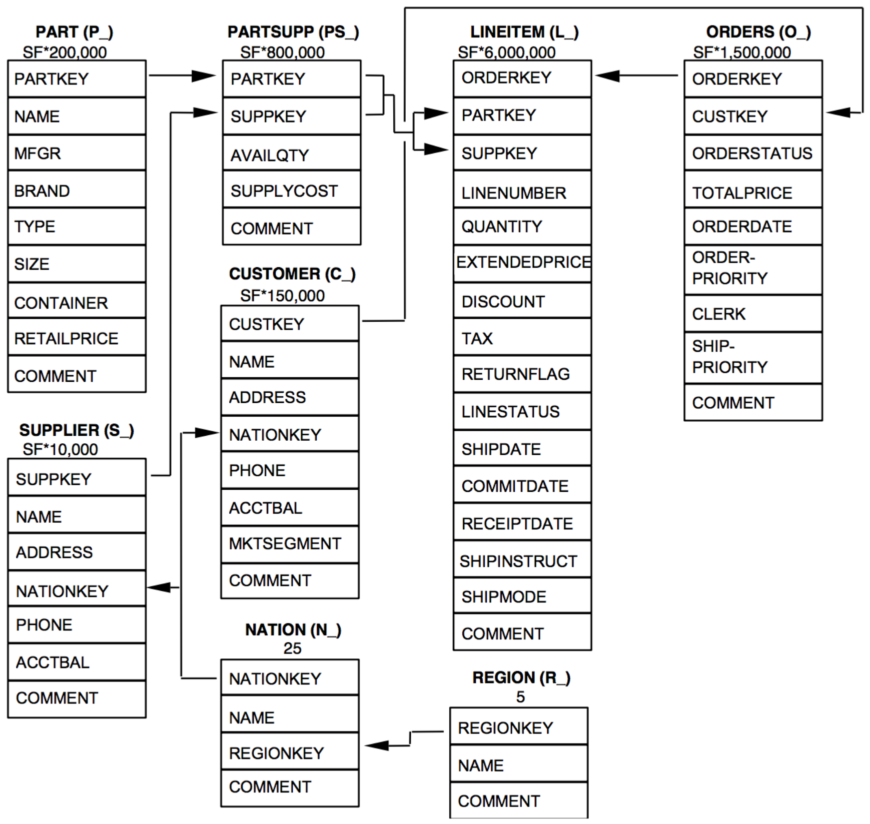
12. [4 points]: At which values of `movies.year` (in the range of 1893 to 2115) do the plans change? Do you believe the query planner is switching at the correct points? (justify your answer quantitatively).

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.

A diagram of the schema of TPC-H is shown in Figure 1.

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.



Legend:

- The parentheses following each table name contain the prefix of the column names for that table;
- The arrows point in the direction of the one-to-many relationships between tables;
- The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the Scale Factor, to obtain the chosen database size. The cardinality for the LINEITEM table is approximate (see Clause 4.2.5).

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=10, the `lineitem` table will have 60 million records, since the figure shows that `lineitem` has size SF*6,000,000.

Consider the following query, representing Query 3 in the TPC-H benchmark, which computes the total revenue from a set of orders in the "BUILDING" market segment placed during a certain date range.

```
SELECT
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = 'BUILDING'
    AND c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate < date '1995-03-15'
    AND l_shipdate > date '1995-03-15'
GROUP BY
    l_orderkey,
    o_orderdate,
    o_shippriority
ORDER BY
    revenue desc,
    o_orderdate
```

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, and a `customer` record is 179 bytes, as outlined in Section 4.2.5 of the TPC-H spec. Thus, an SF=10 `lineitem` table takes 112 * 60M = 6.7 GB of storage.
- All key attributes are 4 bytes, all numbers are 4 bytes, dates are 4 bytes, the `c_mktsegment` field is a 10 byte character string, the `o_shippriority` field is a 15 byte character string (assume strings are fixed length).
- `l_discount` is uniformly random between 0.00 and 1.00.
- `o_orderdate` is selected uniformly random between '1992-01-01' and '1998-12-31' - 151 days.
- `o_shipdate` is uniformly random in (`o_orderdate`, `o_orderdate` + 121 days).
- `c_mktsegment` is selected uniformly and randomly from 'AUTOMOBILE','BUILDING','HOUSEHOLD','FURNITURE', and 'MACHINERY'.
- `o_shippriority` is selected uniformly and randomly from '1-URGENT','2-HIGH','3-MEDIUM','4-NOT SPECIFIED', and '5-LOW'.
- `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 10 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 10 ms, and the disk can sequentially read 100 MB/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 1 GB buffer pool, and an additional 1 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. [3 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. [2 points]: Estimate the runtime of the query in seconds (considering just I/O time).

15. [3 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. [2 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. [2 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

Supposed you are creating a database to keep track of the floors, rooms, and denizens of your dorm / apartment / house / multi-room tent / large domicile.

Specifically, you will need to keep track of:

1. The rooms in your domicile; room have sizes (in ft^2), and names.
2. The denizens of each room; rooms may have 0 or more denizens, and denizens may reside in 0 or more rooms; denizens have first and last names and birthdays.
3. The corridors between rooms; each corridor connects exactly two rooms together.
4. The relationships between denizens; each denizen may have a relationship with zero or more other denizens. Relationships have types (friend, enemy, etc.)

18. [2 points]: Write out a list of functional dependencies for this schema.

19. [3 points]: Draw an ER diagram representing your database. Include a few sentences of justification for why you drew it the way you did.

20. [2 points]: Write out a schema for your database in BCNF. Include a few sentences of justification for why you chose the tables you did.

21. [2 points]: Is your schema redundancy and anomaly free? Justify your answer.

22. [2 points]: Suppose you wanted to ensure that each denizen has more friends than enemies (since otherwise they would be kicked out of the domicile.) How can you enforce this constraint?

23. [2 points]: Suppose you decide that each denizen should reside in exactly one room. How would your schema change? Suggest a view that will allow legacy programs written across the old database to continue to function. Does your view provide full backwards compatibility, i.e., are there any cases that it does not handle properly?