6.830 Problem Set 3 (2018)

**Assigned:** Friday, Oct 5, 2018

**Due:** Monday, Oct 15, 2018, 11:59 PM

Submit to Gradescope: [https://gradescope.com](https://gradescope.com)

The purpose of this problem set is to give you some practice with concepts related to 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 `imdb` database which includes information about movies and people how were involved in those moves. You can use commands such as `\d` and `\d <tablename>` to learn more about the data. We have loaded this data into a Postgres server which you will be using for this assignment. To access the server, you can ssh into your kerberos@athena.dialup.mit.edu and start a session with:

```
psql -h geops.csail.mit.edu 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)
```

If you run `EXPLAIN ANALYZE` in addition to giving statistics and plans, Postgres will actually execute the query.

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 \textit{foo} and \textit{bar} is identified as \textit{(foo, bar)}.

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

2. [2 points]: Which query plan does Postgres choose for \texttt{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: \texttt{select title from movies order by title}.

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

Suppose we want to run the following query.

\begin{verbatim}
select movies.title, ratings.rating from movies, ratings
where movies.id = ratings.movie_id;
\end{verbatim}

5. [2 points]: What type of join does Postgres choose to use for this query? What are 2 other types of joins it could have used, and under what conditions would each one be optimal?
6. [2 points]: Consider the two following queries and their plans from \texttt{imdb}:

```sql
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)
```

```sql
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?
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.

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 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 (modified) statistics about the database:

- A lineitem record is 224 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 224 * 60M = 13.44 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.
- l_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 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 disk seeks take 1 ms, and the disk can sequentially read 500 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 4 GB buffer pool, and an additional 2 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.
7. [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.

8. [3 points]: Estimate the runtime of the query in seconds (considering just I/O time).

9. [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?