CS 166 Lab 9: Optimization

1. Introduction

Optimization is a key part of the database to speed up performance of queries. In a way, it does many things from keeping statistics and creating indexes. In this lab, we will examine how the PostgreSQL optimizer picks the query plan for different queries.

EXPLAIN Clause

The main tool we will be using in this phase is the EXPLAIN clause. If you add this clause before running a query, it will output the query plan used for this query.

For the following examples, the data has the following tables:
Stores(storeid, name, state, latitude, longitude)
Catalog(itemid, itemname)
Sales(storeid, itemid, time, cost)

```
EXPLAIN ANALYZE SELECT * FROM stores;
```

If we run the above expression we will get the query plan chosen by the optimizer. By adding the ANALYZE along with EXPLAIN we will also get the time the optimizer calculates that this query will take.

```
Seq Scan on stores  (cost=0.00..12.90 rows=290 width=250) (actual time=0.021..0.206 rows=1000 loops=1)
 Total runtime: 0.322 ms
```

As you can see, since the actual query asks for all the tuples from Stores the optimizer just uses a sequential scan on the dataset.

```
EXPLAIN ANALYZE SELECT * FROM stores WHERE state='CA';
```

Assume now that we add a WHERE statement, that effectively provides the stores whose state is California. In order to do this, the query needs to filter the data based on the state. Since there is no index, the optimizer still uses the sequential scan; see the plan below. Note that the time projected is slightly more, since there is filtering happening in main memory.

```
Seq Scan on stores  (cost=0.00..20.50 rows=20 width=25) (actual time=0.047..0.290 rows=20 loops=1)
  Filter: ((state)::text = 'CA'::text)
  Rows Removed by Filter: 980
Total runtime: 0.354 ms
```

Note that the query plan is always shown from top down. So in this case, it first does a sequential scan on stores and then filters out tuples so as to get the stores that are in California. Let's now create an INDEX on the state for the stores table called stateIndex. By default, in PostgreSQL this is a B+-tree. Since the file is not ordered according to the Stores attribute, this index is a non-clustered index.

```
CREATE INDEX stateIndex on stores(state);
```

Running the same query from the previous example we get this plan:

```
Bitmap Heap Scan on stores  (cost=4.41..12.66 rows=20 width=25) (actual time=0.160..0.200 rows=20 loops=1)
  Recheck Cond: ((state)::text = 'CA'::text)
  ->  Bitmap Index Scan on stateindex  (cost=0.00..4.40 rows=20 width=0) (actual time=0.146..0.146 rows=20
        loops=1)
      Index Cond: ((state)::text = 'CA'::text)
Total runtime: 0.292 ms
```

Note, that when PostgreSQL considers this query, the index will provide a number of store record ids that have CA, but some of them may still be in the same stores page. Instead of fetching each page separately, it does a “bitmap index scan” which identifies the different pages needed. It then does a “bitmap heap scan” that goes in the original stores file and brings only the pages that contain one or more records with state CA. This happens when there are many results returned by the non-clustered index (many stores that are in CA).

One thing to note is that an INDEX might not be used by the optimizer under the following conditions (see for example here)

1. Can’t use index:
    a. Index doesn’t exist
    b. Query asks for a function computed on the attribute value on which the index is created
    c. Data Type mismatch (due to data type casting)
    d. Index does not support the operator
2. Does not think it will be faster
    a. Table is too small (so scanning may be faster)
    b. Large portion of rows returned (like the example above)
    c. Limit Clause (give the top 10 results) because it can be aborted early

Now let’s consider joins. Below we have a simple join between the stores and sales table so we find out about all the sales for each store. We have to join the storeid otherwise you won’t know which sale corresponds to which store.

```
EXPLAIN ANALYZE SELECT * FROM stores, sales WHERE stores.storeid = sales.storeid;
```

```
Hash Join  (cost=30.50..5301.75 rows=175000 width=45) (actual time=0.595..98.194 rows=175000 loops=1)
  Hash Cond: (sales.storeid = stores.storeid)
  ->  Seq Scan on sales  (cost=0.00..2865.00 rows=175000 width=20) (actual time=0.027..23.066 rows=175000
        loops=1)
  ->  Hash  (cost=18.00..18.00 rows=1000 width=25) (actual time=0.528..0.528 rows=1000 loops=1)
      Buckets: 1024  Batches: 1  Memory Usage: 58kB
      ->  Seq Scan on stores  (cost=0.00..18.00 rows=1000 width=25) (actual time=0.008..0.193 rows=1000
            loops=1)
```

```
Total runtime: 106.564 ms
```

In the above plan, the sales table is the outer relation, while the stores table is the inner relation of the join. The actual join is a hash join, so the plan scans sequentially the stores table and it hashes the stores records (inner) to create a hash table on storeid. Then it goes sequentially through sales and checks if the storeid is in the stores table using the hash table.

```
EXPLAIN ANALYZE SELECT state, COUNT(*)
FROM stores
GROUP BY state;
```

Above we want to get the number of stores in each state. It's a simple group_by and aggregate query.

```
 HashAggregate  (cost=24.00..24.50 rows=50 width=3) (actual time=0.739..0.752 rows=50 loops=1)
   ->  Seq Scan on stores  (cost=0.00..19.00 rows=1000 width=3) (actual time=0.016..0.212 rows=1000 loops=1)
 Total runtime: 0.957 ms
(3 rows)
```

According to the plan, the only difference is that there is a HashAggregate after doing a sequential scan on the stores. A sequential scan is used because all records are needed.

```
EXPLAIN ANALYZE SELECT DISTINCT st.storeid
FROM stores st, sales s
WHERE st.storeid = s.storeid and s.itemid=1 and
      st.storeid in (SELECT st2.storeid
                     FROM stores st2, sales s2
                     WHERE st2.storeid = s2.storeid and
                           s2.itemid=3);
```

Above we have a nested query where we want the stores that sold both item 1 and item 3. In this case we have to join the results of stores that sold item 1 and stores that sold item 3.

```
HashAggregate  (cost=6316.51..6326.51 rows=1000 width=4) (actual time=46.499..46.638 rows=949 loops=1)
  ->  Nested Loop  (cost=3410.24..6307.49 rows=3605 width=4) (actual time=35.864..45.543 rows=3372 loops=1)
        Join Filter: (st.storeid = s.storeid)
        ->  Hash Join  (cost=3410.24..3442.99 rows=1000 width=12) (actual time=35.816..36.427 rows=971
loops=1)
              Hash Cond: (st.storeid = st2.storeid)
              ->  Seq Scan on stores st  (cost=0.00..19.00 rows=1000 width=4) (actual time=0.011..0.151 rows=1000
                    loops=1)
              ->  Hash  (cost=3397.74..3397.74 rows=1000 width=8) (actual time=35.781..35.781 rows=971 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 38kB
                    ->  HashAggregate  (cost=3387.74..3397.74 rows=1000 width=8) (actual time=35.356..35.529
                          rows=971 loops=1)
                          ->  Hash Join  (cost=31.50..3379.47 rows=3307 width=8) (actual time=0.675..34.075 rows=3465
                                loops=1)
                                Hash Cond: (s2.storeid = st2.storeid)
                                ->  Seq Scan on sales s2  (cost=0.00..3302.50 rows=3307 width=4) (actual
                                      time=0.044..31.818 rows=3465 loops=1)
                                      Filter: (itemid = 3)
                                      Rows Removed by Filter: 171535
```

```
                        ->  Hash  (cost=19.00..19.00 rows=1000 width=4) (actual time=0.614..0.614 rows=1000
                              loops=1)
                            Buckets: 1024  Batches: 1  Memory Usage: 36kB
                            ->  Seq Scan on stores st2  (cost=0.00..19.00 rows=1000 width=4) (actual
                                  time=0.006..0.282 rows=1000 loops=1)
      ->  Index Only Scan using sales_pkey on sales s  (cost=0.00..2.81 rows=4 width=4) (actual
                        time=0.006..0.008 rows=3 loops=971)
            Index Cond: ((storeid = s2.storeid) AND (itemid = 1))
            Heap Fetches: 3372
 Total runtime: 46.951 ms
(21 rows)
```

In the plan, the HashAggregate is used to choose the distinct values generated. This is why it is the top operator and it is also used later as part of the plan earlier. Next we have a nested loop that ensures that the stores and sales storeid is the same. Since we have a primary key of (storeid, item), it does use an index scan on the sales table to get the items with itemid 1. On the other hand, there is a Hash Join used to find the stores that sold itemid 3. This plan is similar to a previous example to find the stores that sold itemid 3 with the exception that you filter sales for those with itemid 3.

2. Lab Assignment

For this assignment, you will unzip lab8.zip into your directory. Run the below command to move the data files into your temp folder.

```
cp *.txt /tmp/$USERNAME/mydb/data/
psql -h localhost -p $PORT $USERNAME_DB < schema.sql
psql -h localhost -p $PORT $USERNAME_DB < insert.sql
```

Run the following query files in the following order:

1. Run query.sql
2. Run index.sql
3. Run query.sql

Save the results of all 3 commands and briefly explain the difference between the plans with or without the index.