CSCD43 DATABASE SYSTEM TECHNOLOGY

Due Friday April 3

Weight: 12% of your final grade

Assignment 3

Query Optimization

1 Outline

In this assignment, you will carry out a number of experiments involving the optimization of queries using the PostgreSQL query optimizer. The primary focus will be on **statistics** collected and used by the PostgreSQL query optimizer.

2 Setup

1. Download the files `one.csv` and `two.csv` from

   http://www.cs.toronto.edu/~koudas/courses/cscd43/one.csv
   http://www.cs.toronto.edu/~koudas/courses/cscd43/two.csv

   The tables corresponding to these files can be created by the following SQL statements.

```
CREATE TABLE One(
    id integer,
    b integer,
    c integer,
    d integer);

CREATE TABLE Two(
    id integer,
    a integer);
```

2. Load the `csv` files into the database using the `copy` command of `psql`. Create indices on *each* of the attributes in the database, using the `Create Index` command.

3. Run the `vacuum analyze` command to update the statistics on the tables.

4. Run a few sample queries to get an idea of the ranges and data distributions on each of the attributes.

3 Evaluation

3.1 System Catalogs

Each time you run `vacuum analyze` on PostgreSQL , the system catalogs are updated. The extent of statistics collection is controlled by a variable `default_statistics_target` that can take values between 0 and 1000, with the default value of 10. This variable can be changed using the `set` command of `psql`. *Each time you change the value of this variable, you should run `vacuum analyze` to update the statistics.*
PostgreSQL collects a wide variety of statistics about each attribute. The most important ones are:

- the most common values (`MCVs`) and their frequencies, and

- a set of bucket boundaries for an equi-width histogram.

This statistical information can be easily accessed by simple SQL queries on the PostgreSQL system catalogs. The system catalogs of PostgreSQL are well documented and the statistical information can be easily found with a casual websearch.
Answer the following questions.

1. Use the PostgreSQL catalog to find the number of distinct values of each of the attributes in table *One*. Write the SQL query you used to find out this information.

2. Use the catalog to find out the space used by the relation *Two* on the disk, in number of bytes. Write down the SQL query used to find this information.

3.2 Equality Queries

Consider a set (with size at least two) of equality queries of the form

```
SELECT * FROM One WHERE d = value;
```

1. Set `value` to a high-frequency value of the attribute `d`. Which query plan is selected by PostgreSQL to execute this query? (This information can be found by using the `explain` command).

2. Set `value` to a low-frequency value of the attribute `d`. Which query plan is selected by PostgreSQL to execute this query now?

3. If the two query plans (corresponding to the two cases) are different, try to find the *cross-over* point in terms of frequency, i.e., the point

   - **above** which one plan is used, and
   - **below** which the other plan is used.

   Why does such a cross-over point exist?

3.3 Single Column Range Queries

Generate 10 queries of the form

```
SELECT COUNT(*) FROM One WHERE c > value1 AND c < value2;
```

where (obviously) $value1 < value2$.

1. Using the default statistics, compare the *estimated* and *actual* selectivities of the predicates for each of the 10 queries.

2. Collect more statistics by increasing the `default_statistics_target` value, and running `vacuum analyze` again.

3. Plot the average estimation error over the set of queries, as a function of the `default_statistics_target` value (between 10 and 500).
   The estimation error for a query is computed as follows:

$$\frac{|\texttt{EstimatedCardinality} - \texttt{ActualCardinality}|}{\texttt{ActualCardinality}}$$

   The average estimation error for a set of queries is the average of the estimation error over all the queries in the set.
   For the given attribute (`c`), what do you think is a good value for the `default_statistics_target`, with respect to the average estimation error? (i.e., the value where there should be a point beyond which you get diminishing returns in accuracy). Justify your choice using the graph created above.

3.4 MultiColumn Range Queries

Before starting this section, reset `default_statistics_target` to its default value (10) and run `vacuum analyze`. Consider a set (of size 10) of queries of the form

```
SELECT COUNT(*) FROM One
WHERE c > value1 AND c < value2 AND
      d > value3 AND d < value4;
```

In general, you should find that the optimizer estimates are well off the actual estimates. This is true for queries that return many values, and for queries that return very few or no values.

1. Increase the `default_statistics_target` value as you did in Section 3.3. Plot the average estimation error for the queries in a graph

2. Do the estimates come significantly closer to the actual values? Why or why not? What is the wrong assumption that the optimizer is making? Use information from the graph and/or the `Explain` command to explain why you believe the optimizer is making the assumption you think it is making.

3.5 Simple Joins

As before, reset `default_statistics_target` to its default value (10) and run `vacuum analyze`.
Consider an id-based join of the form

```
SELECT COUNT(*) FROM One, Two
WHERE One.id = Two.id AND
     <some selectivity conditions on One> AND
     <some selectivity conditions on Two>;
```

Answer the following questions:

1. Disable hash-joins and sort-merge joins using the `set` command. The optimizer should now be using a nested-loop join (maybe with an index). Which relation is set as the inner and which one is used as an outer (i.e., the smaller or larger relation)? Try to come up with a query that flips the inner and outer relations when more statistics are available for it. Explain what new information led to this change.

2. Answer the above question when a hash-join is used (instead of a nested-loop one).

3. When does PostgreSQL prefer which join operator (i.e., `nested-loop, hash-join`, or `sort-merge` join)? Generate 100 join queries with random single attribute range selectivity conditions. By "random" we mean uniformly distributed across the space of possible selectivities that a range condition could have. For each query, run the `explain` command to extract the optimizers' choice of join. Make a scatter plot of these queries in 2 dimensions, as described below:

   - The `X`-axis represents the estimated selectivity of the predicate on the inner relation.
   - The `Y`-axis represents the estimated selectivity of the predicate on the outer relation.
   - For each such `(x,y)` point, mark the position on the graph with a symbol corresponding to which join algorithm is used (`nested-loop`, `hash`, or `merge-join`). Can you see any interesting trends? When is which algorithm used? *Note: This question will take forever to complete, if you attempt it by hand. A fairly simple script will make it much easier!*

4. Increase the `default_statistics_target` to a very high number and update the statistics. Run the `explain` command on the 100 queries, as above. How many of the queries had their plans changed (i.e., join operator or inner/outer order)? Give an example of a query which had its operator changed. Explain why.

5. Modify the tables to add a `primary key` constraint on the column `id` for *both* the tables. This can be done using the `Alter Table` command (Look up the documentation to see how). Repeat the experiment you did in Question 3. Did the choice of plans change for many queries? Now add an additional constraint on Table `Two` adding a foreign key constraint on `Two.id` referencing column `One.id`. Again repeat the experiment and see whether this additional information changed any choices made by the optimizer. In both cases provide a scatter plot displaying the choice of the optimizer.

4 Submission instructions

Submit a report (`pdf/ps`) only, containing your answers to all the questions above. Make sure your names/student numbers are on that report.

Good luck!