Lab 8 : CS 387 Jan-Apr 2024

Examining and understanding query plans, query optimization, in postgres

Part-A Preparing the database

1. Create a database called univ8
2. Load the university schema into your database – DDL.sql
3. In case you had loaded the schema + data, and want to start from scratch, you need to drop tables first – DDL-drop.sql
4. Use the psql command line to load file largeRelationsInsertFile.sql . You will see "INSERT 0 1" being printed for every insert. It will take some time but will terminate successfully. You may see an unchanging screen even though the system is continuously printing the above line. Hit enter to see if prints are ongoing.

Part-B Exploring various query plans and query optimizations

In each of the queries below, you can create/delete indices on appropriate relation attributes, as necessary.

1. **query1.sql** : Create a query where PostgreSQL uses bitmap index scan on relation takes. Explain why PostgreSQL may have chosen this plan. Reference: https://www.geeksforgeeks.org/bitmap-indexing-in-dbms/ . Also note that PostgreSQL does not have a specific command for bitmap index creation.
2. **query2.sql** : Create a selection query with an AND of two predicates, whose chosen plan uses an index scan on one of the predicates.
3. **query3.sql** : Create a selection query with an OR of two predicates, whose chosen plan uses an index scan on one or both of the predicates.
4. **query4.sql** : Create a query where PostgreSQL chooses a (plain) index nested loops join. NOTE: PostgreSQL uses nested loops join even for indexed nested loops join. The nested loops operator has 2 children. The first child is the outer input, and it may have an index scan or anything else, that is irrelevant. The second child must have an index scan or bitmap index scan, using an attribute from the first child.

5. Create an index as below, and see the time taken to create the index: create index i1 on takes(id, semester, year);

6. Similarly see how long it takes to drop the above index using: drop index i1;

7. **times.csv** : Create a table takes2 which has the same schema as takes. Insert the first N entries of takes into takes2. Now create index i2 on takes2, similar to index i1 on takes. Measure the time taken for index i2 creation and deletion, as a function of N. Explain your observation. Your CSV file should have 3 columns: N, createTime, dropTime . Have say 5-6 rows for different values of N. Last row of CSV is just a plain text sentence or two with your explanation.

Part-C Continued exploration of various query plans and query optimizations

In each of the queries below, you can create/delete indices on appropriate relation attributes, as necessary. You can assume that the prior part-B queries have been executed before this one (in case you are using some index created earlier). In each SQL file you submit, including the "explain" keyword as appropriate.

1. Create an empty table takes3 with the same schema as takes but no primary keys (it has the foreign keys though). Find the time taken for insertion of the first N entries of takes, into takes3, i.e. to execute the query: insert into takes2 (select * from takes LIMIT <N>). Let us denote this time as insert_without_PK_time(N). Vary N as 3000, 6000, 9000, 12000, 18000, 24000, 30000. To measure the time insert_without_PK_time(N), measure thrice, and take the median value, in milli-seconds. Be sure to start from an empty takes3 table for each measurement. Write a python script for this, using the psycopg library. Filename: **measure_takes3.py**

2. Using psql, find the query plan for the above insert statement. This part is for your own learning – nothing to submit.

3. Next measure the time it takes to modify takes3 by using alter table to add the primary key constraint. You must add the primary key constraint after insertion of N entries into the table. Let us denote this time as add_PK_time(N). Vary N as earlier. You can just modify the earlier measure_takes3.py file, by adding the alter table statement to a suitable place inside the loop varying N. As earlier, your time measurement will be the median of 3 measurements.

4. Now in the same loop, measure the time taken to insert N entries into takes2 (empty to begin with). Remember – takes2 is the same as takes3 except for the primary key. Let us denote this time as insert_with_PK_time(N). As earlier, measurement taken should be the median of 3 runs for each value of N.

5. Enhance the script to plot a graph with X-axis as N. The Y-axis should be the time, in milli-seconds. There should be 4 plots: insert_without_PK_time(N), add_PK_time(N),

insert_with_PK_time(N), and finally insert_plus_add_PK(N) = insert_without_PK_time(N) + add_PK_time(N). Output file: **meas.png**

6. What do you observe in the above graph? Record your observation in a comment line toward the end of file measure_takes3.py, in a single comment line beginning with: **`## OBSERVATION`**

7. **query5.sql** : Create a query where PostgreSQL chooses a merge join (hint: use an order by clause). In this part, do not use takes2 or takes3, but only the original database tables. Also, do not use any "limit" clause here (see next part).

8. **query6.sql** : Add a LIMIT 10 clause to the previous query (this should be the only change in the query), and see what algorithm method is used. (LIMIT n ensures only n rows are output.) Explain what happened, if the join algorithm changes; if the plan does not change, create a different query where the join algorithm changes as a result of adding the LIMIT clause. Add your explanation for why the query plan changes as a comment line toward the end of query6.sql, in a single comment line beginning with: **`-- OBSERVATION`**

9. **query7.sql** : Create an aggregation query where PostgreSQL uses hash-aggregation without any sorting. Here too, do not use takes2 or takes3, but only the original database tables.

10. **query8.sql** : Create an aggregation query where PostgreSQL uses sorting followed by aggregation. Here too, do not use takes2 or takes3, but only the original database tables.