PROJECT PART-2

By- Janaki Raghuram Srungavarapu & Jetha Harsha Sai Emandi

1. About the System

The database system that we used for this project is PostgreSQL. The reason why we chose postgres is:

- We felt it would be easier to compare parameters of the same system than comparing two different Databases that are hosted on different resources. By choosing a single system we intend to keep the other parameters such CPU costs, Operating System and hardware resources the same while we toggle between the internal parameters of the system.
- We used PostgreSQL in our previous Introduction to Database Management class and thus have a fair knowledge of the system. It also has rich Documentation. It is the most advanced open source database management system. Also, the PostgreSQL has a very interactive community which can help us in guiding if we ever run into any difficulties during the project.
- One advantage is that it is robust in nature with high performance and multitasking. Scaling is another advantage in PostgreSQL. Since it is an open source database management system, there are a lot of active users who develop and propose various new modules to the community.
- Postgress gives immense flexibility to play around various memory parameters and Planner Cost Constants which is much needed for this project.
- These arrangement parameters can be used to alter the query plans picked by the query optimizer and drive the optimizer to pick an alternate arrangement. Better approaches to improve the nature of the plans picked by the optimizer incorporate changing the planer cost constants.

2. System Research

We have looked at various memory and query planning parameters of postgres and decided to consider the following parameters for the purpose of benchmarking.

- **Work_mem:**

  This memory component indicates the measure of memory to be utilized by internal sort activities and hash tables creations. The default work_mem is postgres is set to 4 MB. Sort operations are used for ORDER BY, DISTINCT queries. It is also used for computation of hash tables. Hash tables are utilized in hash joins, hash-based accumulation, and hash-based processing of IN subqueries. Increasing or decreasing these parameters affects the performance of sort/hash queries.

  There could be multiple sort/hash operations in a single query. There could be multiple queries running in parallel. Work_mem is the amount of memory that each of these sort/hash operations could use. Hence setting up large work memory could result in bad performance when there are too many parallel transactions.

- **Enable_seqscan:**

  Sequential scan is one of the basicscan operations of the database for retrieving data on disk. Sequential scan is always conceivable. Regardless of the schema of the table or the presence of indices on the table, sequential scan always returns data. This parameter cannot be totally turned off. Setting this parameter off only encourages query optimizer to use other types of scans such as index and bitmap. By default sequential scan is on.

  Sometimes it is easy to sequentially scan the pages from disk rather than retrieving pages based on indexes or bitmap scan. If a large percent of the table is returned by a query, a sequential scan is usually always the better option to use because a sequential scan performs sequential I/O whereas the other alternatives available most probably will perform random I/O.

- **Enable_hashjoin**:

  Hash join is one of the most frequently used join algorithms by postgres. It is primarily due to its efficiency to produce results at less cost compared to other joins. However, it can only handle equi joins.

  Hash join is mostly used on tables that are not already sorted on the join columns. The optimizer builds an in-memory hash table on the inner tables join column using work_mem . The optimizer then scans the outer table for matches to the hash table. The

cost of performing a hash join highly depends on whether or not the hash table fits in the memory. Cost rises significantly if the hash table has to be written to disk. We intend to turn this join off and see what joins the optimizer picks. And compare the execution times of hash join against the other joins.

- **Shared_buffers:**

  This is the actual amount of memory allocated to postgresql page queue. The default memory that postgres sets is 128MB.This could be configured to higher or lower sizes based on the available RAM and other OS requirements. Based on my research so far, it is best practice to limit the shared buffer size to less than ¼ of the RAM.

  Based on the average RAM size today, the deafult 128 MB seems to be far below the optimal value. We plan to test various query execution times varying the shared buffer. Shared buffer can only be set during the server restart.

- **Enable_mergejoin:**

  Enables or disables the query planners use of merge-join plan types. The default is on. If the query planner thinks the hash table required for a hash join will surpass memory, postgres may consider using merge join. If the the relationship is already sorted on the join attributes, merge join should theoretically perform better than hash join. While toggling between hash join, we would like to observe its impact on merge join.

3. Performance Experiment Design:

Performance Experiment 1:

- **Performance Issue:**

  This explores the issue of how the execution time varies according to the join algorithms.

- **Data set:**

  The dataset used here consists of two relations. One relation consists of five thousand tuples and another relation contains fifty thousand tuples.

- **Query:**

  The query performs an inner join between the two relations fivektup and fiftyktup, consists of five thousand and fifty thousand tuples. It selects the attribute string4 of fivektup relation.

  ```
  set enable_hashjoin = off;
  select fivektup.string4 from fivektup, fiftyktup1
  where fivektup.unique2 = fiftyktup1.unique2
  and fiftyktup1.unique2 < 1000;
  ```

- **Parameters:**

  The parameter that we would be toggling here is enabling or disabling the hash_join.

- **Results Expected:**

  As the number of tuples in one table is less which will fit in the memory, it will opt for hash join, which might provide the best execution time. Now, if we turn off hash join, the execution time increases. It uses merge join.

Performance Experiment 2:

- **Performance Issue:**

  In this experiment, we explore how the parameter of sequential scan has an effect on the query execution runtime.

- **Data set:**

  The dataset that we use here has just one tuple fivektup which consists of five thousand tuples. But we also make use of one more relation fiftyktup2 which consists of fifty thousand tuples. This relation is used inside the EXISTS clause.

- **Query:**

  In this query, we just extract the twenty attributes from the fivektup relation. We test whether it exists in the subquery.

```
set enable_seqscan = off;
explain analyze select f1.twenty from fivektup f1
where f1.twenty in
(select f2.twenty from fiftyktup2 f2
where f2.unique1 between 250 and 750 and f1.two = f2.two);
```

- **Parameters:**

  The parameter that we are experimenting with here is enable_seqscan. We first run the query using seqscan, then we toggle it off and run the query again.

- **Results Expected:**

  According to us, if we run the query using the sequential scan, it may in less amount of time. But we think if we toggle it off, the query can take longer time than it does with sequential scan.

Performance Experiment 3:

- **Performance Issue:**

  This test explores the relation betweens the work_mem and the sort operation on the relationships. This could also be used to test the performance of hash joins under varying work memory.

- **Data set:**

  Three different relationships of size 10k, 100k and 1000k tuples are used in this performance test.

- **Query:**

  The query takes tuples that are not indexed and sorts them in the order of string4. If time permits, hashing of relation in work memory for the hash table generation will also be tested against the varying work memory.

  set work_mem = varied (1 MB to 128 MB);

  SELECT unique2, two, string4 FROM *fiftyktup* ORDER by string4;

  SELECT fiftyktup.unique2, hundredktup.unique2 FROM fiftyktup, hundredktup
  WHERE fiftyktup.string4 = hundredktup.string4;

- **Parameters:**

  The parameter that we would be toggling here is work_mem.

- **Results Expected:**

  As the sorting and hashing happens in the work memory. We are expecting improved performance with increased work memory when dealing with larger relations that don't fit in work memory.

Performance Experiment 4:

- **Performance Issue:**

  This test explores the relation betweens shared_buffers and the execution time of large queries whose relations do not fit in memory.

- **Data set:**

  Three different relationships of size 100k and 1000k tuples are used in this performance test.

- **Query:**

  A non indexed query joins two relations where their string4 match. The same query is tested with varying relationships and varying relation sizes for the following cases

  - At least one relationship fits in memory
  - No relationship fits in memory

  SELECT fiftyktup.unique2, hundredktup.unique2 FROM fiftyktup, hundredktup
  WHERE fiftyktup.string4 = hundredktup.string4;

- **Parameters:**

  The parameter that we would be toggling here is Shared_buffere size. The size of the relationships could also be altered to gain more information.

- **Results Expected:**

  Theoretically. The join algorithms should give better execution times with increased buffer sizes.

4. Lesson Learned:

This part of the project taught us to know about the different kinds of configuration parameters that can be used to test the database performance. Mainly, we got an in-depth knowledge of how the runtime of the query is affected just by making a small change in the configuration parameter. The issue that we faced in this part is creating queries that would make us understand the effect of the change in parameters. Ultimately, it gave us a thorough understanding of how a wide range of sql queries can be used to test the performance of the database system.

We also realised that when an experiment is run twice with the same table while toggling a parameter, the pages of the relation could end up in the buffer pool after the first experiment leading to inaccurate results during the second experiment. We learn that it is important to flush the pages of the bufferpool to obtain more accurate comparisons. We learnt how to access pg_ buffercache and check the tables existing in the shared_buffers.

```
CREATE EXTENSION pg_buffercache;
```

```
dbimpl=# select c.relname,pg_size_pretty(count(*) * 8192) as buffered,
dbimpl-#         round(100.0 * count(*) / (
dbimpl(#            select setting from pg_settings
dbimpl(#            where name='shared_buffers')::integer,1)
dbimpl-#         as buffer_percent,
dbimpl-#         round(100.0*count(*)*8192 / pg_table_size(c.oid),1) as percent_of_relation
dbimpl-# from pg_class c inner join pg_buffercache b on b.relfilenode = c.relfilenode inner
dbimpl-# join pg_database d on ( b.reldatabase =d.oid and d.datname =current_database())
dbimpl-# group by c.oid,c.relname order by 3 desc limit 10;
       relname         | buffered | buffer_percent | percent_of_relation
-----------------------+----------+----------------+---------------------
onemilliontup_unique3  | 24 MB    |           18.5 |                79.3
onemilliontup_unique1  | 24 MB    |           18.5 |                79.3
hundredktup            | 23 MB    |           17.7 |                95.5
fiftyktup              | 12 MB    |            9.3 |                99.8
onemilliontup_stringu1 | 4936 kB  |            3.8 |                 5.0
onemilliontup          | 3904 kB  |            3.0 |                 1.6
onemilliontup_stringu2 | 1224 kB  |            0.9 |                 1.6
pg_depend              | 472 kB   |            0.4 |                95.2
onemilliontup_unique2  | 368 kB   |            0.3 |                 1.7
pg_description         | 328 kB   |            0.3 |                91.1
(10 rows)


dbimpl=#
```

While studying about work_mem and shared_buffers, we also learnt the impact of kernel buffer/OS cache might have on the execution times. Sometimes a file must be flushed out of the shared buffer but sit in the OS cache making its access easier than assumed.