PROJECT PART - 2

By: Pratik Kadam & Samarth Kedilaya

1. Which option / system (s) you will be working with and why you chose it (them)

Solution: The database system that we used for this project is PostgreSQL. The reason why we chose Postgres is given below:

- It is the most advanced open source database management system. And also, we were already quite familiar with it as we also used it in the CS 586 class. It helps in giving us enterprise level performance and functions compared to any other open source database management systems with limitless possibilities. Also, the PostgreSQL has a very interactive community which can help us in guiding if we ever run into any difficulties during the project.
- Scaling in 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
- PostgreSQL has a great deal of ability. It underpins complex structures and a broadness of implicit and client characterized information types. It gives broad information limit and is trusted for its information honesty. You may not require the majority of the propelled highlights we've surveyed here for putting away information, however since information needs can develop rapidly, there is without a doubt, clear advantage to having everything readily available.
- These arrangement parameters give an unrefined strategy for impacting the query plans picked by the query optimizer. On the off chance that the default plan picked by the optimizer for a specific query isn't ideal, a transitory arrangement is to utilize one of these setup parameters to 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 or running ANALYZE physically.

2. System Research (30 pts)

Solution: These are the following parameters that we would be using:

- **Enable_mergejoin:**

  Enables or disables the query planner's use of merge-join plan types. The default is on. At the point when Postgres thinks the hash table required for a hash join will surpass memory, Postgres will ordinarily utilize a merge join. A merge join ends up utilizing plate substantially more successfully than a hash join can. Like a hash join, a merge join just takes a shot at joins where the join condition is an equity channel.

- **Work_mem:**

  Indicates the measure of memory to be utilized by internal sort activities and hash tables before writing to impermanent plate records. Note that for a mind boggling inquiry, a few sort or hash activities may keep running in parallel; every task will be permitted to use as much memory as this esteem determines before it begins to compose information into brief records. Likewise, a few running sessions could be doing such activities simultaneously. Along these lines, the all out memory utilized could be ordinarily the estimation of work_mem; it is important to remember this reality while choosing the esteem. Sort tasks are utilized for ORDER BY, DISTINCT, and combine joins. Hash tables are utilized in hash joins, hash-based accumulation, and hash-based processing of IN subqueries.

- **enable_seqscan:**

  Enables or disables the query planner's use of sequential scan plan types. It is impossible to suppress sequential scans entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is on. There are two primary reasons why Postgres will execute sequential scans:

  The first is that sequential scan is always conceivable. Regardless of what the schema of the table is, or what indices are on the table, Postgres always has the alternative of running a sequential scan.

  The other principle reason is that at times a sequential scan is in reality quicker than other existing different choices. When reading data from a disk, reading data sequentially is generally quicker than reading the data in a random order. 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_bitmapscan:**

  Enables or disables the query planner's use of bitmap-scan plan types. The default is on.A bitmap index scan is a method for joining more than one index. A bitmap index scan works by utilizing the 1st index to find the majority of the rows that fulfill the 1st filter, then utilizing the 2nd filter to find all indexes that fulfill the 2nd filter, after which results are combined to obtain the locations of all rows in the table that fulfill the above two mentioned filters. Later, Postgres will get the rows from the physical table in the order the rows are in the physical table.

  Getting the rows in the physical order from the table that are in the table has it own advantages. In the event that it turns out an enormous part of the table is returned, a bitmap scan turns out to be fundamentally the same as a sequential scan. In specific situations, this can even prompt a bitmap index scan being superior to an ordinary index scan in the event when just a single index is utilized. However, the drawback of this is that Postgres loses knowledge about the sort order.

- **enable_nestloop :**

  Enables or disables the query planner's use of nested-loop join plans. It is impossible to suppress nested-loop joins entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is on.

  Even though nested loops are comparatively inefficient, they do have one significant advantage. A nested loop is the only join algorithm that Postgres has that can be utilized to evaluate any join. Irrespective of what the join condition is and what indexes are present, Postgres always has the choice of executing a nested loop.

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. Both the relations has twenty lakh tuples.

- **Query:**
  The query performs an inner join between the two relations twentylakh1 and twentylakh2, both consisting of twenty lakh tuples.It also uses the DISTINCT keyword on the attribute string4 of twentylakh1 relation.

  ```
  SET enable_mergejoin = off;

  SELECT DISTINCT twentylakh1.string4 FROM twentylakh1, twentylakh2
  WHERE (twentylakh1.unique2 = twentylakh2.unique2)
  AND (twentylakh2.unique2 < 1000)
  ```

- **Parameters:**
  The parameter that we would be toggling here is enabling or disabling the merge_join.

- **Results expected:**
  As the number of tuples in either of the tables will not fit in the memory, it will opt for merge join, which might provide the best execution time.
  Now, if we turn off merge join, the execution time increases. It uses nested loop.

Performance Experiment 2:

- **Performance issue:**
  This experiment explores how the query execution time varies when we vary the the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files. It

- **Data set:**
  The dataset used here consists of two relations. Both the relations has twenty lakh tuples.

- **Query:**
  This query selects the unique1 attribute of relation ‘twentylakh1’. It performs an inner join between the two tables on unique1 and unique2. It make sures that the data that is retrieved has the value of stringu1 attribute starting from ‘AAAB’.

  ```
  SET work_mem = 4MB;
  SET work_mem = 128MB;

  SELECT twentylakh1.unique1 as MINU1
  FROM twentylakh1, twentylakh2
  WHERE twentylakh1.unique1 = twentylakh2.unique1 AND
  twentylakh1.unique2 = twentylakh2.unique2 AND twentylakh1.stringu1 LIKE
  'AAAB%'
  ```

- **Parameters:**
  The parameter here we are toggling here is work_mem. At first we used the value as 4MB. Then we try using work_mem of 128MB.

- **Results expected:**
  The work_mem parameter has a direct affect on the execution time of the query. When the work_mem is low, the query executes lower. When the work_mem is high, the query executes faster.

Performance Experiment 3:

- **Performance issue:**
  In this experiment, we analyse how the query gets affected if we stop using the bitmap scan in postgres.

- **Data set:**
  The dataset that we use here has just one tuple ‘tenlakh1’ which consists of ten lakh tuples. But we also make use of one more relation ‘twentylakh2’ which consists of twenty lakh tuples.

- **Query:**
  Here, we extract the ‘twenty’ attribute from tenlakh1 relation. In the subquery, we extract ‘ten’ attribute from twentylakh2. Within the subquery, we use the BETWEEN clause for specifying the range of values. The unique1 attribute has a range of 2499-7499. onepercent attribute has a values between 25 and 75.

  ```
  SET enable_bitmapscan = 'off';

  EXPLAIN ANALYZE SELECT tl1.twenty from tenlakh1 tl1
  WHERE EXISTS (select distinct tl2.ten from twentylakh2 tl2
                where tl2.unique1 BETWEEN 2499 and 7499 and
                tl2.onepercent BETWEEN 25 AND 75 and tl1.two = tl2.two );
  ```

- **Parameters:**
  In this experiment, we first run the query without setting any parameters. So, it uses bitmap scan. Then, we toggle the bitmapscan off and run the query again.

- **Results expected:**
  According to us, the query runs faster when the bitmap scan is being used. When we run the query after turning bitmap scan off, the query takes more time to execute. The query then makes use of the parallel sequential scan.

Performance Experiment 4:

- **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 ‘tenlakh1’ which consists of ten lakh tuples. But we also make use of one more relation ‘twentylakh2’ which consists of twenty lakh tuples. This relation is used inside the EXISTS clause.

- **Query:**
  In this query, we just extract the ‘twenty’ attribute from the tenlakh1 relation. We test whether it exists in the subquery.

  ```
  SET enable_seqscan = 'off';

  SELECT tl1.twenty from tenlakh1 tl1
  Where EXISTS (select distinct tl2.ten from twentylakh2 tl2 where tl2.unique1
  BETWEEN 250 and 750 and tl1.two = tl2.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.

4. Lessons Learned

This lead to an expanded comprehension of database framework execution and the effects of database usage on execution. We came across 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 mainly 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 how a wide range of sql queries can be used to test the performance of the database system.