CS 487/587 Database Implementation
Winter 2021
Database Benchmarking Project - Part 2

Sowmya Sri Indukuri, Sukanya Kothapally

About the System:

- The System we are working on is PostgreSQL. The reason to choose it is as we are familiar with it and have experience working on it from previous terms. We also feel that it has a super friendly user interface and convenient to use.
- Apart from the above reason there are many features PostgreSQL supports which is also a reason to choose it.
- PostgreSQL comes with many features which helps developers build applications, administrators to protect data integrity and build fault-tolerant environments, and helps to manage the data no matter how big or small the dataset.
- PostgreSQL provides advanced locking mechanisms, tablespaces, partitioned tables, and many different types of indices. It can also run parallel queries and offers advanced cost-based query optimization.
- PostgreSQL is designed to be extensible. PostgreSQL allows you to define your own data types, index types, functional languages, etc.If you don't like any part of the system, you can always develop a custom plugin to enhance it to meet your requirements e.g., adding a new optimizer.
- PostgreSQL has many system and query parameters, which can be adjusted according to the data being fetched and this is very much useful in this project.

System Research:

Work_mem:

Work mem is a Postgres configuration that specifies the amount of memory that can be used for certain operations. The configuration of work mem appears easy at its surface. Work mem only defines the amount of memory available for internal sorting processes and hash tables before entering data on disk. However, leaving work_mem unsettled can lead to a number of problems. What is even more disturbing is that you get a memory out of your database error and leap into tuning work_mem, only to act unintelligently.

The default work mem value in Postgres is 4MB, which is probably a little less. This means that per Postgres activity like joins, some sorts, etc. can consume 4MB before it starts spilling to disk. When Postgres starts writing temp files to disk, obviously things will be much slower than in memory. You can find out if you're spilling to disk by searching for temporary files within your PostgreSQL logs when you have log_temp_files enabled. If you see a temporary file, it can be worth increasing your work_mem.

Enable_hashjoin:

Value given to this parameter basically enables or disables the query planner's use of hash-join plan types. If enable_hashjoin is set to False, the cost of conducting a HashJoin operation will increase so that it will not appear in an execution plan. PostgreSQL is more likely to pick a Nested Loop or MergeJoin operator over enable_hashjoin when enable_hashjoin is off.

The strategy of hash join in PostgreSQL is, it sequentially scans the internal relationship and generates a hash table, which includes all the connecting keys used by the = operator. Then it sequentially scans the outer relation and samples the Hash for each row that matches the link keys. This is a bit similar to nested loop connection. The development of the hash table represents a more start-up effort, but it is much quicker to test the hash than to scan an inner relationship.

Default: on

Enable_seqscan:

Enables or disables the query planner's use of sequential scan plan types. The Seq Scan operation scans the entire relation (table) as stored on disk. 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.

To execute a sequential scan, Postgres literally iterates through a table a row at a time and returns the rows requested in the query which is not efficient in cases where the tables are large.There are two main reasons why Postgres will execute sequential scans. The first is that a sequential scan is always possible. No matter what the schema of the table is, or what indexes exist on the table, Postgres always has the option of executing a sequential scan.The other main reason is that in some cases a sequential scan is actually faster than the other options available. When reading data from disk, reading data sequentially is usually faster than reading the data in a random order. If a large portion of the table will be returned by a query, a sequential scan usually winds up being the best way to execute it. This is because a sequential scan performs sequential I/O whereas the other options available mostly perform random I/O.

Default: on

Enable_mergejoin:

Enables or disables the query planner's use of merge-join plan types. Each relation is sorted on the join attributes before the join starts. Then the two relations are scanned in parallel, and matching rows are combined to form join rows. This kind of join is more attractive because each relation has to be scanned only once. The required sorting might be achieved either by an

explicit sort step, or by scanning the relation in the proper order using an index on the join key. Merge Joins are preferred if the join condition uses an equality operator and both sides of the join are large, but can be sorted on the join condition efficiently. Disabling merge join to false makes the postgres to either select hash join or nested loop join.

Default: on

Enable_indexscan:
Enables or disables the query planner's use of index-scan plan types. In an index scan, the index access method is responsible for repeating all the tuples it has been told about that match the scan keys. Index scan uses different data structures (depending on the type of index) corresponding to the index involved in the query and locates required data (as per predicate) clause with very minimal scans. Then the entry found using the index scan points directly to data in the heap area, which is then fetched to check visibility as per the isolation level. Index scan should be chosen only if overall gain outperforms the overhead incurred because of Random I/O cost.

Default: on

Performance Experiments:

Performance Experiment 1:

Work_mem: Evaluates the performance of the query planner when given different values for work_mem.

Data sets: 1 HUNDREDKTUP Table

Values considered for work_mem in this experiment:

1. 64kB
2. 4MB
3. 256MB

Queries:
As we know that work_mem is used for internal sort operations and hash tables like ORDER BY, DISTINCT, and merge joins, we are considering one query with DISTINCT and another query with ORDER BY

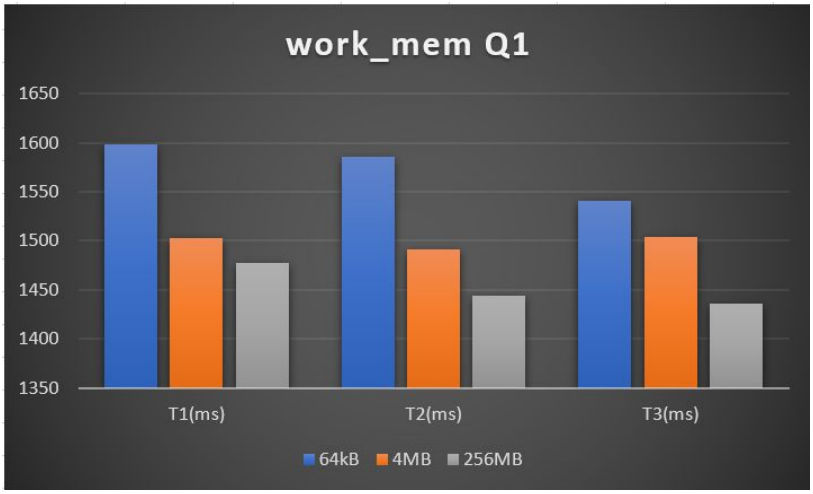
Query1 :

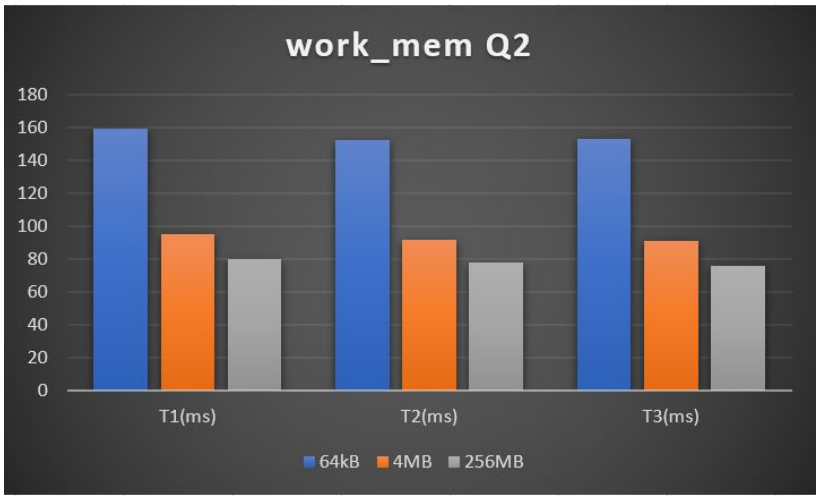
```
explain analyze select distinct u.stringu4  from HUNDREDKTUP u, HUNDREDKTUP
v where u.two = v.two and u.unique3 < 100 and v.unique3 >1
```

Query2 :

```
explain analyze select  two,four,ten from HUNDREDKTUP order by string4;
```

Expected Result:





From the results we can clearly say that, for a query with a big relation and which has some internal sorting operation and with 64kB of work_mem took long execution time and with 256MB of work_mem took performed well with least execution time.

Performance Experiment 2:

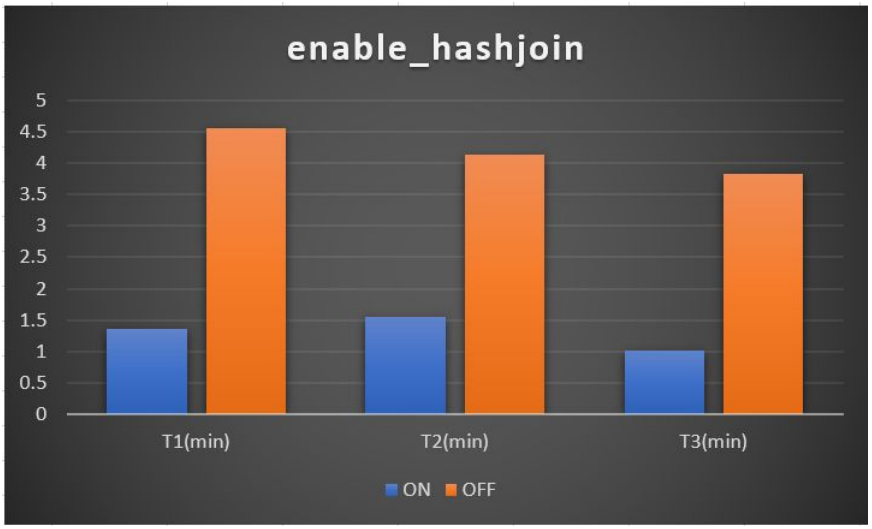
Enable_hashjoin: Here we toggle with enable_hashjoin parameter and evaluate the performance.

Datasets: One relation with 10k tuples and another with 1 Lakh tuples.

Query:

```
explain analyze select * from TENKTUP u, HUNDREDKTUP v where u.two =  v. two;
```

Results:



Hash Join works best when we have a small relation that can fit into the memory and another relation which is extremely big and that doesn't fit into the memory. When the enable_hashjoin is on there is significantly better performance than when it is set off. When hash join is disabled, postgres is more likely to use nestedloopjoin or mergejoin operator and hence taking more time.

Performance Experiment 3:

Enable_seqscan: Enables or disables the query planner's use of sequential scan plan types.

Performance issue: In general to execute a sequential scan, Postgres literally iterates through a table a row at a time and returns the rows requested in the query which is not efficient in cases where the tables are large. In this experiment we are going to analyze the runtime of the query when the parameter is on and off.

Data sets: HUNDREDKTUP , TENKTUP

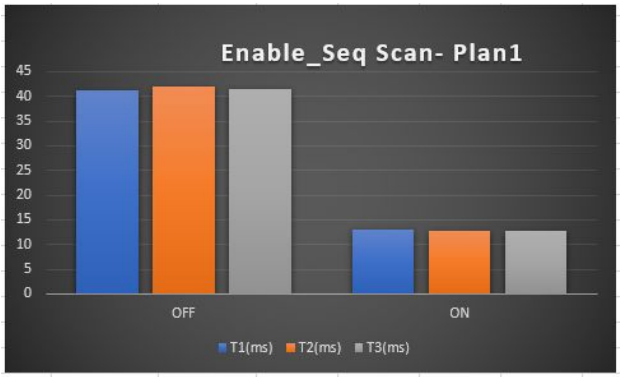
Queries: In the below plans the reason to choose seq scan on unique2 is because from the generated tables, unique2 is sequential(1,2…) and performing seq scan on unique2 gives us correct interpretation. The selectivity factor is 10%;

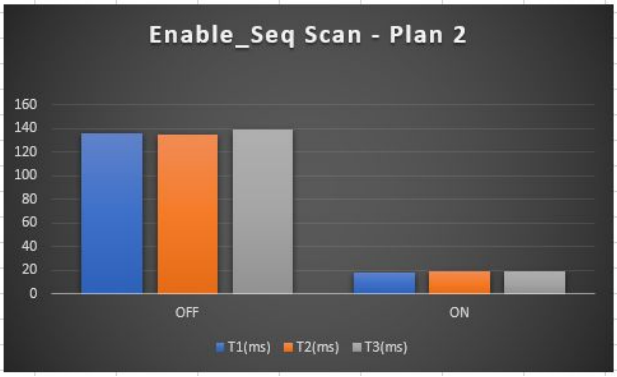
Case 1:

```
set enable_seqscan = off;
explain analyze select unique1,unique2,stringu4 from "HUNDREDKTUP" where unique2 <
10000;
```

Case 2:

```
set enable_seqscan = off;
explain analyze select u.stringu4 from "TENKTUP" u where u.unique2 in (select
v.unique2 from "HUNDREDKTUP" v where v.unique2 < 10000) ;
```





Results: As per the behaviour of seq scan it should iterate through the table a row at a time and then returns the rows requested in the query. In case1, when single table is considered and the seq scan is off ,query planner picks the rows in random order at a time and matches the where condition to pick the rows requested in the query but this is not efficient in the case where the condition mentioned for the column is sequential.In this scenario if the seq scan is on, as the column in the where condition unique 2 is sequential, it filters from the condition and iterates over the rows sequential, thus reducing the run time of the query. Similarly in case 2 when the table sizes are different and there is a nested query, it first selects sequential scan irrespective of the nested loop. But if the sequential scan is off it takes more time as it completely executes the query with a nested loop.

Conclusion:
Run time of the queries is significantly less when the seq scan is on.

Performance Experiment 4:

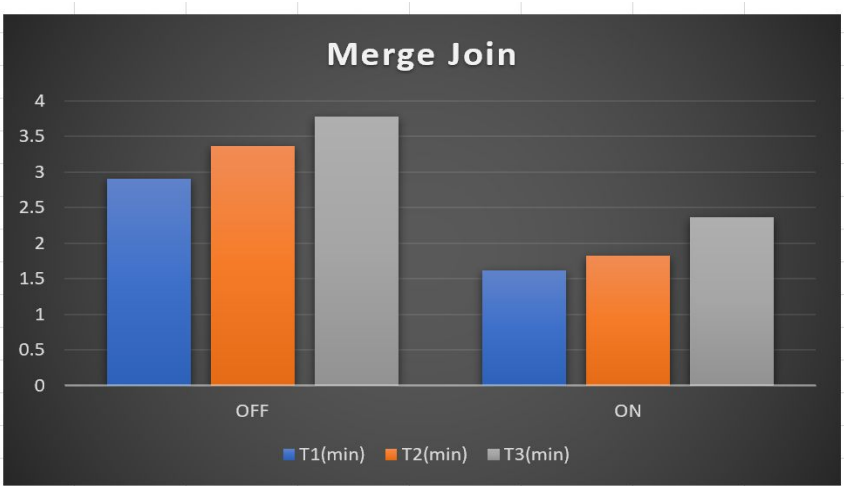
Enable_mergejoin: Enables or disables the query planner's use of merge-join plan types.

Performance issue: In general to execute a join there must be two relations, these two relations are scanned in parallel, and matching rows are combined to form join rows. This kind of join is chosen more because each relation has to be scanned only once.

Data set : TENKTUP

Query:

```
Set Enable_mergejoin=OFF;
Explain analyze Select u.string4 from "TENKTUP" u, "TENKTUP" v where u.two = v.two
and u.two < 1000;
```



Results:

Merge Join is an algorithm wherein each record of outer relation is matched with each record of inner relation until there is a possibility of join clause matching. This join algorithm is only used if both relations are sorted and the join clause operator is “=”. Since both the tables in the query have the same size, postgres should use Merge join . when we set enable_mergejoin parameter to on and find the time the query planner uses the merge join algorithm. When it is set to off it should use another algorithm (maybe nested loop join or hash join) which takes more time. In the experiment when the merge join is set to off the query took more time than when it was on.

Conclusion:
Run time of the query is less when the merge join is on.

Lesson Learned:

- Before doing these experiments we were not aware of how query planner configures its parameters to reduce the time and increase the performance.
- Understood how postgres query optimizer will identify which plan to choose by looking into the cost of each plan.
- Sequential scan is not suppressed completely even when it is off, it runs in the background.

- Initially we started with small data sets and then ended up with large datasets for true analysis.
- We also learnt that higher work_mem value may make complex queries faster if it allows us to fit all the temp data for the query into memory.With increase of work_mem we can check if we are getting a cheaper hash join. However, one should use it carefully because sometimes increasing the value too much may cause out of memory errors on your database server.
- Join strategies are crucial to understand execution plans and tune queries.

References:

- https://www.citusdata.com/blog/2018/06/12/configuring-work-mem-on-postgres/
- https://stackoverflow.com/questions/49023821/nested-join-vs-merge-join-vs-hash-join-in-postgresql
- https://malisper.me/postgres-sequential-scans/
- https://severalnines.com/database-blog/overview-various-scan-methods-postgresql
- https://www.cybertec-postgresql.com/en/join-strategies-and-performance-in-postgresql/
- https://severalnines.com/database-blog/overview-join-methods-postgresql