CS 487/587 Database Implementation
Database Benchmarking Project - Part II

Neha Agrawal
Geetha Madhuri Chittuluri

Relational Database System
PostgreSQL

System we will be working with - PostgreSQL database. There are many reasons for choosing this database. Some of them are -

- PostgreSQL database is quite easy to use. We are familiar and have experience working with this database system. Also the database can handle a lot of data where nested and complex data structures can be created, stored and retrieved.
- PostgreSQL database is an object relational database, which supports user-defined objects and their actions such as data types, indexes, functions, operators and domains - makes this database flexible and robust.
- It is trusted for it's data integrity.. Primary keys, cascading foreign keys, unique, not null, check constraints and other data integrity features ensure only validated data is stored.
- Material views are another virtual table feature supported by PostgreSQL. They can be indexed and unlike regular views, materialized views are snapshots in time.
- PostgreSQL offers index capabilities, supports partial, expression indexes. Built-in support for regular B-tree and hash indexes.
- PostgreSQL has synchronous replication that utilizes two database instances running simultaneously where master database is synchronized with slave database. With synchronous replication, each write waits till the confirmation is received from both master and slave.

From the above inexhaustive list of various features such as data types, data integrity, concurrency, performance, reliability, security, extensibility, and on the top of all, it is an open-source which makes us chose this system.

System Research

Work_mem -

work_mem is a memory configuration parameter that determines how much memory can be used during certain operations. Work_mem specifies the amount of memory available to be used by internal sort operations and hash tables before writing data to disk. For a complex query, several operations might be running in parallel, each operation uses as much memory as the work_mem value specifies, also several running sessions could be doing such operations concurrently. If the sorting or hash table needs more memory than permitted by work_mem, then PostgreSQL will use temp files on disk to perform such operations. Since disk IO is much slower than memory IO, heavy queries may become very slow as the number of tuples in the dataset grows, even though the query may use indexes efficiently. Use the SET command to change work_mem value.

enable_hashjoin -

Enables or disables the query planner's use of hash-join plan types.

Default value: True

Modify time: SET command

Override: SET enable_hashjoin to [true/false]

Setting enable_hashjoin to false makes it more likely that PostgreSQL will choose NestedLoop or MergeJoin operation over a HashJoin operation. Hash join is usually the fastest. Hash joins only work where the join condition is an equality condition. This works in two phases. Build phase – a hash table is built using the inner relation records. The hash key is calculated based on the join clause key, Probe phase – an outer relation record is hashed based on the join clause key to find a matching entry in the hash table. The case when a hash join won't be preferred is when postgres thinks the hash table needed for the hash join won't fit in memory.

enable_mergejoin -

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

Default value: True

Modify time: SET command

Override: SET enable_mergejoin to [true/false]

Setting enable_mergejoin to false makes it more likely that PostgreSQL will choose NestedLoop or HashJoin operation over a MergeJoin operation. 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 is only used if both relations are sorted and the join clause operator is “=”. So, In order to choose the Merge Join plan, it needs to first sort all records retrieved from both tables and then apply the merge join, which leads to the cost of sorting will be additional and hence the overall cost will increase.

enable_nestloop -

Enables or disables the query planner’s use of nested loop-join plan types.

Default value: True

Modify time: SET command

Override: SET enable_nestloop to [true/false]

Setting enable_nestloop to false makes it more likely that PostgreSQL will choose HashJoin or MergeJoin operation over a Nested Loop Join operation. As setting false increases the estimated cost of performing a nested loop operation so that it is not likely to appear in an execution plan. Nested loop join is the simplest join algorithm wherein each record of outer relation is matched with each record of inner relation. It is the most common joining method which can be used almost on any dataset with type of join clause. Since, the algorithm scans all tuples of inner and outer relation, it is considered to be the most costly join operation. Some of the queries may not have a join clause, in that case, only choice to join is nested loop join.

enable_seqscan -

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

Default value: True

Modify time: SET command

Override: SET enable_seqscan to [true/false]

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.

As an example, If the SELECT returns more than approximately 5-10% of all rows in the table, a sequential scan is much faster, as a sequential scan only requires a single IO

for each row - or even less because a block (page) on the disk contains more than one row, so more than one row can be fetched with a single IO operation. Said that, for smaller tables sequential scan is always faster, but sometimes due to invalid parameters, postgreSQL skips the indexes and starts scanning the table which leads to poor performance. In this situation select to disable the sequential scan option so that query planner is forced to use available indexes and execute the query.

Performance Experiment Design

Experiment 1

This Experiment compares the performance of no index vs clustered vs unclustered index on 100k tuples with 5%, 10%, 20%, 50% selectivity.

Dataset : We will use one relation ‘hundredktup’ with 100k tuples.

Queries : In this experiment, we will be varying the queries to check the elapsed time and query plan used by the postgres. We will run 10 variants of each of the following Queries. The response time will be measured by computing the average elapsed time of all the 10 equivalent queries.

10% selectivity: (Only one variant of each query is shown below.)

- **No Index :** *Insert into temp Select * from db.hundredktup where unique3 between 1081 and 11080*.
- **Clustered Index :**
  - First create a clustered index on unique2 :

    create index idx_clustered on db.hundredktup(unique2)

  - Then check the elapsed time of the following query with clustered index.

    *Insert into temp Select * from db.hundredktup where unique2 between 392 and 10391.*

- **UnClustered Index :**
  - First create an unclustered index on unique1 :

    create index idx_unclustered on db.hundredktup(unique1)

  - Then check the elapsed time of the following query with an unclustered index.

*Insert into temp Select * from db.hundredktup where unique1 between 62951 and 72950.*

Similarly, we will test for 5%, 20% and 50% selectivity.

Parameters Varied : In this experiment, we won't vary system parameters or scale the dataset sizes. Instead, we will run various queries with clustered, nonclustered or no index to compare the performance of the postgres system.

Expected Result:

I. Elapsed time

Suppose elapsed time for a query without index, clustered and nonclustered index be $T_{noIndex}$, $T_{cluster}$ and $T_{uncluster}$ respectively.

Selectivity	No Index	Clustered	Unclustered
5%, 10%	$T_{noIndex} > T_{cluster}$ $T_{noIndex} > T_{uncluster}$	$T_{cluster} < T_{noIndex}$ $T_{cluster} < T_{uncluster}$	$T_{cluster} < T_{uncluster} < T_{noIndex}$
20%, 50%	$T_{noIndex} > T_{cluster}$ $T_{noIndex} \simeq T_{uncluster}$	$T_{cluster} < T_{noIndex}$ $T_{cluster} < T_{uncluster}$	$T_{uncluster} > T_{cluster}$ $T_{uncluster} \simeq T_{noIndex}$

As shown in the table above, for selectivity less than or equal to 10%, queries with clustered and unclustered indices are expected to outperform the queries without indices.

For selectivity greater than 10%, unclustered indices does not improve the performance of the queries and its execution time is comparable with no index queries whereas clustered index is still expected to execute faster than unclustered and no index queries.

II. Query Plan by Postgres

Selectivity	No Index	Clustered	Unclustered
5%, 10%	Sequential Scan	Index Scan	Index scan
20%, 50%	Sequential Scan	Index Scan	Sequential Scan

As shown in the table above, for selectivity less than or equal to 10%, it is expected that postgres will perform sequential scan on queries without indices and index scan on queries with clustered and unclustered indices.

However, for selectivity greater than 10%, index scan on queries with unclustered index won't improve performance and instead postgres is expected to go for sequential scan rather than index scan. It continues to use index scan for clustered indexed query and seq scan for no-index query.

Experiment 2

In this experiment we intend to explore equi-join queries which generally uses Sort-Merge join or Hash join algorithms. We will vary the relation size and check what join algorithm is used by postgres. Moreover, we'll toggle the system parameters - enable_hashjoin, enable_mergejoin and compare the execution runtime.

We'll Consider following three scenarios:

Scenario 1: At Least one of the relations fits into the memory.

- **Dataset : (**100k $\times$ 10k)
  A is a big relation with 100k tuples and B is a small relation with 10k tuples.
- **Query :** We will run 4 variants of the following Query. The response time will be measured by computing the average elapsed time of all the 4 equivalent queries. Since one of the relations will fit into the memory then it is expected that postgres will choose a hash join plan. We will toggle the enable_hashjoin parameter to check what other query plan does postgres use and compare their execution time. We will take Selectivity factor as 1%

  Set enable_hashjoin = on; (default)

  Set enable_hashjoin = off;

  Select t1.unique2, t2.stringu1 from db.hundredktup t1, db.tenktup1 t2 where t1.unique1 = t2.unique1 and t1.unique1 < 1000;

- **Parameters :** We will toggle the enable_hashjoin parameter to check what other query plan does postgres use and compare their execution time.
- **Expected Result :** Since relation "tenktup" will fit into the memory then it is expected that postgres will choose a hash join plan. We will disable the enable_hashjoin parameter which by default in on and expect that any other

join algorithm (probably sort merge join plan) will cost more than the hash join plan.

Scenario 2: There is a huge difference in size of two relations A and B. (None of them fits into the memory)

- **Dataset : (**5lakh × 20k)
  A is a huge relation with 5 lakh tuples and B is relatively small relation with 20k tuples. Selectivity factor is 0.5%
- **Query :** Since there is a huge difference in size of 2 relations, it is expected that postgres will choose a hash join plan. We will toggle the enable_hashjoin parameter to check what other query plan does postgres use and compare their execution time. We will take Selectivity factor as 0.5%.

  Set enable_hashjoin = on; (default)

  Set enable_hashjoin = off;

  Select t1.unique2, t2.unique1 from db.twentylakhtup t1, db.twentyktup1 t2 where t1.unique1 = t2.unique1 and t1.unique1 between 50001 and 70000;

- **Parameters :** We will toggle the enable_hashjoin parameter to check what other query plan does postgres use and compare their execution time.
- **Expected Result :** Since there is a huge difference in size of 2 relations it is expected that postgres will choose a hash join plan. We will disable the enable_hashjoin parameter which by default in on and expect that any other join algorithm will cost more than the hash join plan.

Scenario 3: Both the relations are of comparable size. (None of them fits into the memory)

- **Dataset : (**100k × 100k)
  Both A and B are the same size relations consisting of 100k tuples each.
- **Query :** Since both the relations are of comparable size, it is expected that postgres will choose a sortMerge join plan. We will toggle the enable_mergejoin parameter to check what other query plan does postgres use and compare their execution time. We will Selectivity factor as 10%

  Set enable_mergejoin = on; (default)

  Set enable_mergejoin = off;

  Select t1.unique2, t2.stringu1 from db.hundredktup t1, db.hundredktup1 t2 where t1.unique1 = t2.unique1 and t1.unique1 < 10000;

- **Parameters :** We will toggle the enable_mergejoin parameter to check what other query plan does postgres use and compare their execution time.

- **Expected Result :** Since both the relations are of comparable size, it is expected that postgres will choose a sortMerge join plan. We will disable the enable_mergejoin parameter which by default in on and expect that any other join algorithm (probably hash join plan) will cost more than the sort merge join plan.

Summary of Expected Result in all the three scenarios:

Scenario	Parameter	Query Plan	Execution Time Analysis
Rel A fits into mem	Set enable_hashjoin =on	Hash Join (T_{hash})	$T_{other} > T_{hash}$
	Set enable_hashjoin =off	Other Join (T_{other})	
A >> B, Neither A nor B fits into mem	Set enable_hashjoin =on	Hash Join (T_{hash})	$T_{other} > T_{hash}$
	Set enable_hashjoin =off	Other Join (T_{other})	
A ≈ B, Neither A nor B fits into mem	Set enable_mergejoin =on	SortMergeJoin(T_{sort})	$T_{other} > T_{sort}$
	Set enable_mergejoin =off	Other Join (T_{other})	

Experiment 3

In this experiment we intend to explore non equi-join queries for which postgres is left with only one choice i.e nested loop joins. We will vary the work memory size and check the execution runtime. Moreover, we'll toggle the system parameters - enable_nestloop and check how the postgres reacts on that.

Dataset : We will use two relations both with 100k tuples.

Queries : In this experiment, we will be varying the work_mem to check the elapsed time of the non equi join queries. We will run 4 variants of the following Query. The response time will be measured by computing the average elapsed time of all the 4 equivalent queries.

Case 1 : Default work memory space: *Set work_mem = '4MB';*

Case 2 : Shrinked memory space: *Set work_mem = '64kB';*

Case 3 : Expanded memory space: *Set work_mem = '128MB';*

In all the above cases, after setting the work_mem parameter, we will run the following queries

Query1 : explain analyze select distinct t1.unique2, t2.unique1 from db.hundredktup t1, db.hundredktup t2 where t1.unique1 < t2.unique1 and t1.unique1<1000 and t2.unique1<1000;

Query2 : explain analyze Select t3.unique1, t2.unique3 from db.fiftyktup1 t3, db.fiftyktup t2 where t3.unique1 < t2.unique1 and t3.unique1 between 10000 and 19000;

Parameters Varied : In this experiment, we will keep the relation size constant. Instead, we will vary the system parameter work_mem and check the performance of postgres. Moreover, we'll disable the enable_nestloop parameter to check what error does postgres throws.

Expected Result:

As shown in the table below, with shrinked memory space (64kB), we expect postgres to take longest execution time, whereas with expanded memory space (128MB), it should take least execution time.

Also, we expect postgres to throw some error when enable_nestloop is set to "off".

Summary of Expected Result in all the three cases:

Cases	Execution Time	Query Plan with enable_nestloop = 'off'
Work_mem = '64kB'	$T_{64kB} > T_{4MB}$ & T_{128MB}	Postgres should throw some error
Work_mem = '4MB'	$T_{64kB} < T_{4MB} < T_{128MB}$	Postgres should throw some error
Work_mem = '128MB'	$T_{128MB} < T_{4MB}$ & T_{64kB}	Postgres should throw some error

Experiment 4

In this experiment we intend to compare the performance of Index Nested Loop join and Block Nested Loop Join in queries with index on join predicate and without index on join predicate. Moreover, we will compare the cost of probing index on smaller relation vs larger relation. Here we will disable the hashjoin and sortmerge join parameters.

Set enable_hashjoin = off;
Set enable_mergejoin = off;

We'll Consider following two scenarios:

Scenario 1: Index and without index on join predicate

- **Dataset : (**1k × 100k)
  A is a small relation with 1k tuples and B is a large relation with 100k tuples.

- **Query :** We will first run the below query without creating an index and then we will run the query with the index**.**

  Create index idx1 on db.onektup(unique1);

  Select t3.unique1, t2.unique3 from db.onektup t3, db.tenlakhtup t2 where t3.unique1 = t2.unique1 and t2.unique1 <95000;

- **Parameters :** We will compare the execution time of block nested loop vs index nested loop with and without index on unique1 of outer relation.
- **Expected Result :** Index on join predicate should make the query faster. Thus execution time of index nested loop is expected to be far better than the block nested loop join

Scenario 2: Index on smaller relation vs index on larger relation

- **Dataset : (**1k × 10lakh)
  A is a small relation with 1k tuples and B is a large relation with 100k tuples

- **Query :**

  Case 1 : *Create index idx1 on db.onektup(unique1);*

  Case 2 : *Create index idx2 on db.tenlakhtup(unique1);*

  Select t3.unique1, t2.unique3 from db.onektup t3, db.tenlakhtup t2 where t3.unique1 = t2.unique1 and t2.unique1 <95000;

- **Parameters :** Here we will vary the query. First we will create index on smaller relation and then we will create index on larger relation and then compare the execution time in both cases.
- **Expected Result :** We expect index on larger relation to improve the total elapsed time.

  Cost of Index Nested loop join = M + M * p_r * cost of probing index

  Here M is the number of pages of relation without index and p_r is the # of tuples per page of that relation.

  Index on smaller relation should increase the cost.

Summary of Expected Result in the above scenarios:

Scenario	Parameter	Query Plan	Execution Time Analysis
Index vs No-index	No index	BL nested loop join	$T_{noindex} > T_{index}$
	Create index idx on onektup(unique1)	Index nested loop join	
Idx on smaller vs idx on larger relation	Create index idx_onek on onektup(unique1)	Index nested loop join.	$T_{idx1k} > T_{idx10lakh}$
	Create index idx_10lakh on tenlakhtup(unique1)	Index nested loop join	

Lessons Learned

- Tuning postgreSQL database using configuration parameters for better performance. Exhaustive knowledge on memory and query planner configuration parameters.
- In the process of tuning, we have explored various system parameters which can possibly improved the execution time of the queries.
- The experiments helped us to explore various join algorithms and understand how a join algorithm outperforms in one scenario whereas in another scenario it miserably fails.
- We got a detailed idea of how the query optimizer decides which query plan to use by comparing the total cost incurred in different cases.

Issues faced:

- The only issue we faced was to come up with appropriate queries for the above four experiments.
- Experimenting with size of relations to understand the performance impact was little difficult especially while performing hash join and sort-merge join operations.
- Expected result was also difficult to predict in some scenarios but in the next project submission we will present a detailed explanation of how weirdly postgres behaved in some cases.

Overall it was a great learning experience. In order to know the objective of the experiments and expected result, you should be very thorough concept wise. For example: when is it beneficial to use block nested loop and when index nested loop will be a great win over Block nested loop join. When does sequential scan outperforms the unclustered indexed queries. How does selectivity affect the performance of the queries?

Questions like : When does the system choose hash join over sort merge join?

Why does the relative position of inner relation and outer relation matter when we talk about index nested loop join whereas in hash join we don’t pay much attention to the relative positioning of inner and outer relations.

This project was a great opportunity for us to deep dive into each concept and answer these questions.

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.