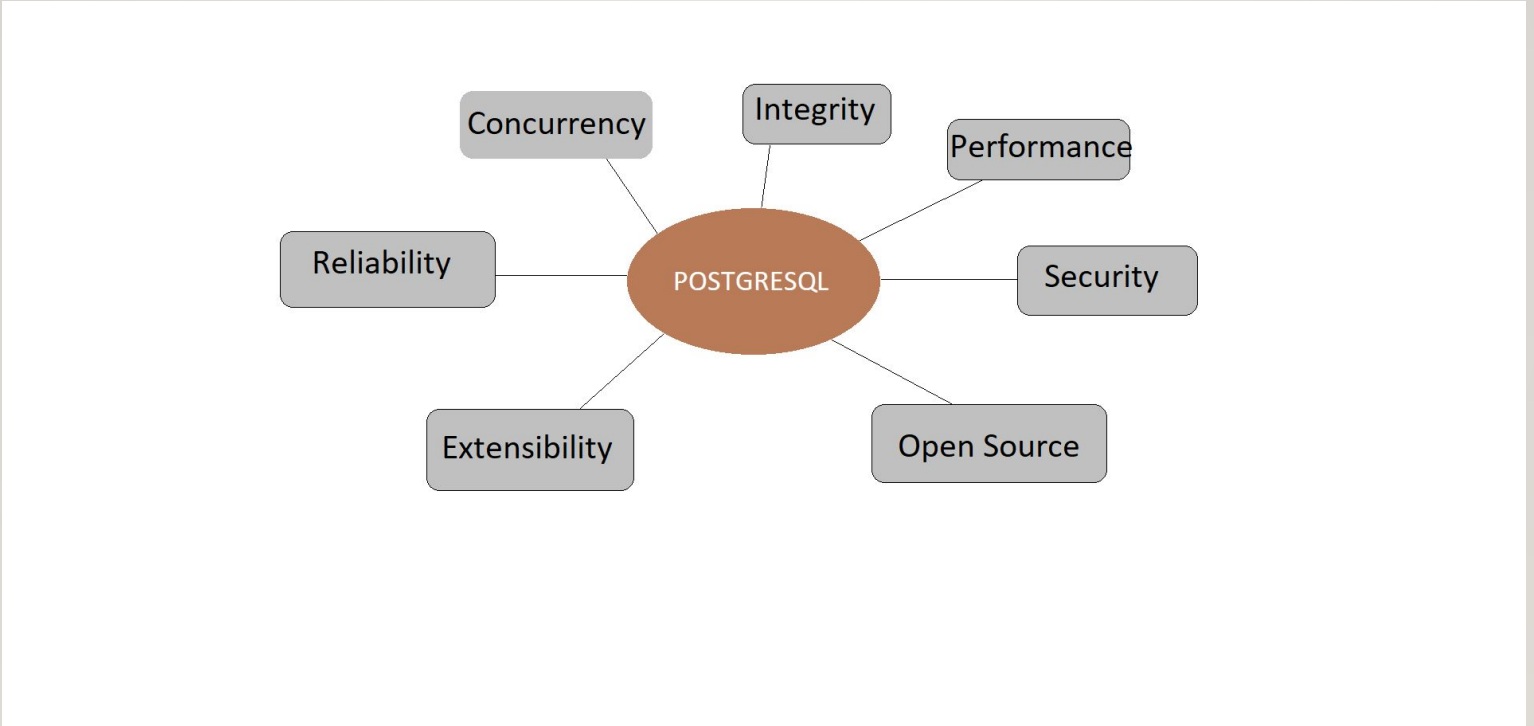
CS587 DATABASE IMPLEMENTATION

Database Benchmarking Project - Part III

- Neha Agrawal
- Geetha Madhuri Chittuluri

DATABASE PREFERENCE AND WHY?

PostgreSQL Version 10.11



WHY POSTGRES?

- Object relational database, which supports user defined objects and their actions.
- Quite easy to use. We are familiar working with this database system.
- Database can handle a lot of data where nested and complex data structures can be created, stored and retrieved.
- Trusted for it's data integrity. Offers index capabilities, supports partial indexes, built-in support for regular B-tree and hash indexes.

GOALS AND APPROACH

Goals –

- To learn about query optimizer and how to tune the PostgreSQL configuration parameters for better performance on query.
- To learn how to make the benchmark scalable.
- To learn about different join algorithms and understand their role in database performance

Approach–

- Varied the relation sizes:
  Relation with 1K, 10K, 50k,100k, 1M tuples
- Varied system memory parameter :
  work_mem used by hash and internal sort operation.
- Toggled various join parameters:
  enable_nestloop, enable_hashjoin, enable_mergejoin
- Created clustered and non-clustered index on selection predicate and join predicate.

Experiment 1

Objective : **Compare the performance of no index vs clustered vs non-clustered index on 100k tuples with 5%, 10%, 20%, 50% selectivity.**

Query used (10% selectivity**):** No Index : ***Insert into temp Select * from db.hundredktup where unique3 between 1081 and 11080.***
Clustered Index : ***Insert into temp Select * from db.hundredktup where unique2 between 62951 and 72950.***
Unclustered Index : ***insert into temp Select * from db.hundredktup where unique1 between 62951 and 72950***

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

1. Elapsed Time:

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.

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%	$T_{noIndex} > T_{cluster}$ $T_{noIndex} \simeq T_{uncluster}$	$T_{cluster} < T_{noIndex}$ $T_{cluster} < T_{uncluster}$	$T_{uncluster} > T_{cluster}$ $T_{uncluster} \simeq T_{noIndex}$

2. Query Plan by Postgres:

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.

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

Experiment 1

Objective : **Compare the performance of no index vs clustered vs non-clustered index on 100k tuples with 5%, 10%, 20%, 50% selectivity.**

Actual Result Found:

	Performance Metric	5%	10%	20%	50%
No Index	Elapsed Time(ms)	113	130	184	704
	Query Plan	Seq Scan	Seq Scan	Seq scan	Seq Scan
Clustered Index	Elapsed Time(ms)	37	67	124	430
	Query Plan	Index Scan	Index Scan	Index Scan	Index scan
Unclustered Index	Elapsed Time(ms)	57.3	127	162	604
	Query Plan	Index Scan	Index Scan	**Index Scan**	Seq Scan

Opposite to the thumb rule: According to the thumb rule, if we query more than 10% of the relation, optimizer goes for seq scan rather than index scan.

- As opposed to the expected behavior, for 20% selectivity optimizer opts for index scan rather than seq scan.
- No index(184 ms) ≅ Uncluster index(162 ms). Although execution time is approximately same in both the cases, still unclustered index has brought down the query execution time by some amount.
- Explanation for this weird behavior: Having index on a relation gives optimizer a head start to go for index scan, and since 20% selectivity is not huge so we assume that optimizer goes with the same logic and prefers for index scan.
- But we can see for 50% selectivity with unclustered index, optimizer opts seq scan over index scan as expected.

Experiment 11

Objective : Explore Hash-join and Sort merge join by varying relation sizes and toggling join parameters and check what join algorithm is used by postgres.

Scenario 1: At least one relation fits into the memory.(100k * 10k)

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

Scenario 2 : There is a huge difference in size of two relations A and B. (None of them fits into the memory) (1000k * 20k)

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

Scenario 3 : Both the relations are of comparable size and requires sorting operation on join attribute. (None of them fits into the memory) (100k * 100k)

```
Select t1.unique2, t2.stringu1 from db.tenlakhtup t1, db.tenlakhtup
t2 where t1.unique2 = t2.unique2 and t1.unique1 between 23000
and 400000 order by t1.unique2;
```

Expected Output:

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 11

Objective : Explore Hash-join and Sort merge join by varying relation sizes and toggling join parameters and check what join algorithm is used by postgres.

Actual Output:

Scenario 1: At least one relation fits into the memory.(100k*10k)

As expected Hash join will cost less and postgres also chooses the same. We can force the optimizer to choose other join algorithm by disabling hash join and we found that postgres opts sort-merge join which increases the overall execution time of the query.

Scenario 2 : There is a huge difference in size of two relations A and B. (None of them fits into the memory) (1000k * 20k)

As expected Hash join will cost less and postgres also chooses the same. When we set enable_hashjoin = off, optimizer opts for merge join and increases the overall execution time of the query.

Scenario 3 : Both the relations are of comparable size and requires sorting operation on join attribute. (None of them fits into the memory) (100k * 100k)

As expected Sort merge join will cost less and postgres also chooses the same. When we set enable_mergejoin = off, optimizer opts for hash join which increase the overall execution time of the query.

Scenario	Parameter	Query Plan and Elapsed Time	Execution Time Analysis
Rel A fits into mem	Set enable_hashjoin =on	T_{hash} = 45 ms	$T_{sort} > T_{hash}$
	Set enable_hashjoin =off	T_{sort} = 74 ms	
A >> B, Neither A nor B fits into mem	Set enable_hashjoin =on	T_{hash} = 170 ms	$T_{sort} > T_{hash}$
	Set enable_hashjoin =off	T_{sort} = 224 ms	
A ≈ B, Neither A nor B fits into mem	Set enable_mergejoin =on	T_{sort} = 85.7 ms	$T_{hash} > T_{sort}$
	Set enable_mergejoin =off	T_{hash} = 194 ms	

Experiment III

Objective : **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.**

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';***

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;*

Expected Output

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 III

Objective : **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.**

Actual Output:

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;*

Cases	Execution Time	Query Plan with enable_nestloop = 'off'
Work_mem = '64kB'	T_{64kB} = 359ms	Block Nested Loop
Work_mem = '4MB'	T_{4MB} = 208ms	Block Nested Loop
Work_mem = '128MB'	T_{128MB} = 129ms	Block Nested Loop

Since elapsed time in all the cases is in ms, so we can't say in concrete manner that expanding work mem would improve the performance of the query.

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;*

Cases	Execution Time	Query Plan with enable_nestloop = 'off'
Work_mem = '64kB'	T_{64kB} = 152 s	Block Nested Loop
Work_mem = '4MB'	T_{4MB} = 97 s	Block Nested Loop
Work_mem = '128MB'	T_{128MB} = 77 s	Block Nested Loop

Here we saw significant improvement in the performance of query with expanded work_mem size.

Additionally, even after disabling the parameter nest-loop, we still see that optimizer chooses nested loop. The good explanation for this behavior is, since postgres has no other option left so it will overlook the configuration set by the user.

Experiment IV

Objective : 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 hash join and sort-merge join parameters.

Scenario 1: Index and without index on join predicate (1K * 100K)

Create index idx1 on db.onektup(unique1);

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

Scenario 2: Index on smaller relation vs index on larger relation (1K * 1000K)

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

Expected Output:

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	

Experiment IV

Objective : **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.**

Actual Output:

Scenario 1: Index and without index on join predicate (1K * 100K) *Same as Expected*

Index on join predicate should make the query faster. Thus execution time of index nested loop is far better than the block nested loop join as expected.

Scenario 2: Index on smaller relation vs index on larger relation (1K * 1000K) *Same as Expected*

Although execution time of these queries came less than 1 sec, but still we see improvement in performance of query when we have index on larger relation. Theoretically, cost of the index loop also showcases same result:

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

Where M is # of pages of relation without index and pr is # of tuples per page of that relation

Scenario	Parameter	Query Plan	Elapsed Time	Execution Time Analysis
Index vs No-index	No index	BL nested loop join	$T_{noindex}$ =14s	$T_{noindex} > T_{index}$
	Create index idx on onektup(unique1)	Index nested loop join	T_{index} = 550ms	
Idx on smaller vs idx on larger relation	Create index idx_onek on onektup(unique1)	Index nested loop join.	T_{idx1k} = 665ms	$T_{idx1k} > T_{idx1M}$
	Create index idx_10lakh on tenlakhtup(unique1)	Index nested loop join	T_{idx1M} = 22ms	

Conclusions

After performing the four experiments and tweaking and playing around various system configuration parameters, we conclude that:

- Queries with Index on selection predicate allows the database server's execution time much faster than it can be without an index.
- However as opposed to the conventional norm of 10% thumb rule, with 20% selectivity of the query and having unclustered index on selection predicate, we found that optimizer have chosen index scan over seq scan.
- We found that of all join algorithms, Hash Join costs less and is mostly preferred by Postgres over other join algorithms and considerably improves the execution time.
- Increase in work memory size, results in faster execution of the queries.
- When working with index nested loop join, having index on join attribute of larger relation drastically improves the execution time of the query.
- For non-equi join queries, even if we disable nested join parameter, system overlook the configuration set by user and continues to use nest-loop join as it's query plan.

Lessons Learned

- Came across various PostgreSQL database configuration parameters for better performance of the queries.
- After exploring no index, clustered and non-clustered indexes, our knowledge has expanded on how postgres performs on various scenarios.
- Exploring on equi-join and non-equi join queries, gave an in-depth understanding on all join algorithms and how the performance got affected after toggling various join parameters.
- Having a small outer relation, we get a big win with index nested loop join over block nested loop join.
- Inner and outer relation make sense for index nested loop join. However, for hash join this doesn't make sense at all.