



# CS 587- Database Implementation Winter 2021 Database Benchmarking Project

Team: Harie Vashini, Zeba Khan Rafi

# System chosen : PostgreSQL

- Strong relational database with easier syntax
- Supports Python language for ease of use
- Has an efficient query optimizer
- Robust in nature with high performance and multitasking
- Provides the benefit of scaling
- An open source database management system with rich documentation
- The GUI tool Pgadmin makes it easy to write and execute queries

# Project Goals

- To learn and evaluate different join algorithms
- To learn about the memory parameter `work_mem` and its effect on queries
- Test the performance of different aggregations
- Index Scan vs Sequential Scan with selectivity

## Approach

Version: PostgreSQL 12.5

### Tables

1. Fivehundredktup : 500,000 tuples
2. Fivemilliontup: 5,000,000 tuples
3. Thousandktup1, Thousandktup2: 1,000,000 tuples

# Experiment 1: Join Algorithms

## Query 1 :

```
SELECT unique1 FROM THOUSANDKTUP1 WHERE unique1 in (SELECT unique1 FROM FIVEHUNDREDKTUP)
```

**Parameter Tested :** enable\_hashjoin


**Objective :** To evaluate the execution time of the query with hash join enabled and disabled.

**Expected Results :** Selectivity is more than 10% hence the query optimizer is expected to choose merge join when compared to hash join as the tables are too large to fit into the memory. When this parameter is disabled, we expect the optimizer to choose merge join.

# Snapshots


enable\_hashjoin = On

Query Editor	
1	SET enable_hashjoin=On;
2	EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
3	SELECT unique1
4	FROM THOUSANDKTUP1 WHERE unique1 in
5	(SELECT unique1 FROM FIVEHUNDREDKTUP)
6	

Data Output	
	QUERY PLAN 
1	Hash Join (actual rows=500000 loops=1)
2	Hash Cond: (thousandktup1.unique1 = fivehundredktup.unique1)
3	-> Seq Scan on thousandktup1 (actual rows=1000000 loops=1)
4	-> Hash (actual rows=500000 loops=1)
5	Buckets: 131072 Batches: 8 Memory Usage: 3221kB
6	-> Seq Scan on fivehundredktup (actual rows=500000 loops=1)
7	Planning Time: 0.240 ms
8	Execution Time: 736.170 ms

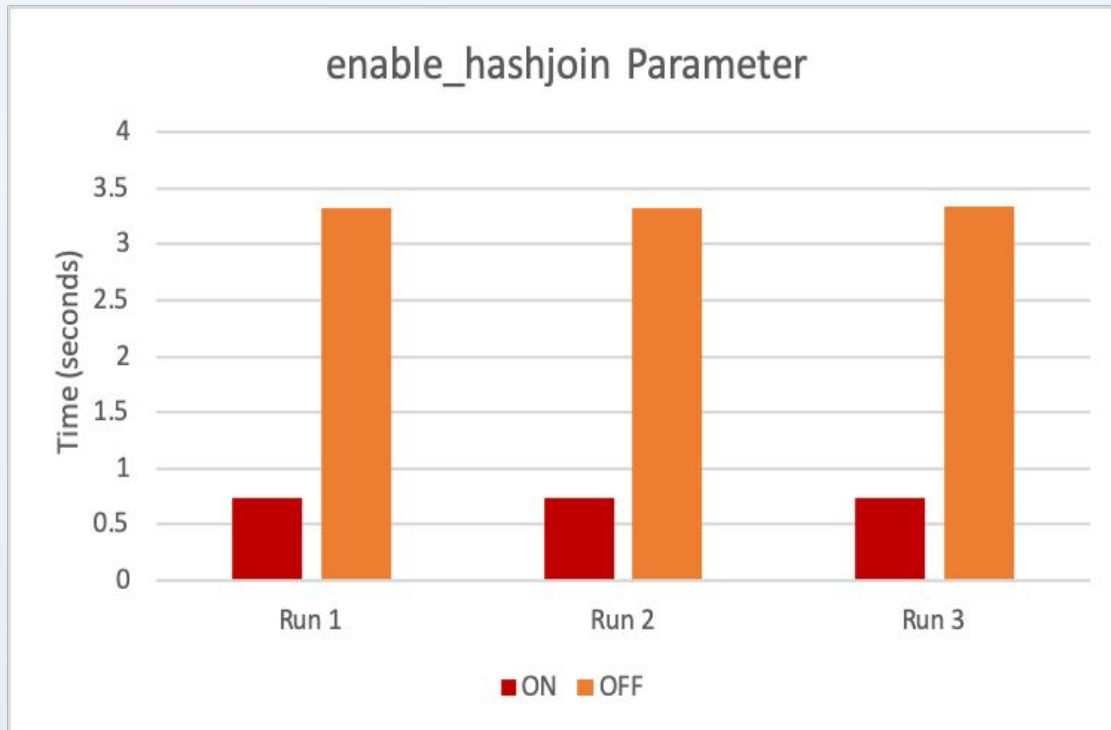
enable\_hashjoin = off

Query Editor	
1	SET enable_hashjoin=Off;
2	EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
3	SELECT unique1
4	FROM THOUSANDKTUP1 WHERE unique1 in
5	(SELECT unique1 FROM FIVEHUNDREDKTUP)
6	

Data Output	
	QUERY PLAN 
1	Merge Join (actual rows=500000 loops=1)
2	Merge Cond: (thousandktup1.unique1 = fivehundredktup.unique1)
3	-> Index Only Scan using thousandktup1_unique1_key on thousandktup...
4	Heap Fetches: 500001
5	-> Index Only Scan using fivehundredktup_unique1_key on fivehundredkt...
6	Heap Fetches: 500000
7	Planning Time: 0.343 ms
8	Execution Time: 3310.549 ms



# Experiment 1: Join Algorithms Results



Average Execution time with  
enable\_hashjoin enabled = 0.7374 s

Average Execution time with  
enable\_hashjoin disabled = 3.3299 s

## Results :

When the hash join parameter is enabled, hash join is used to execute this query which is not expected. The optimizer chooses multi batch hash join splitting the tables into the batches of the same size as the work\_mem and then applying hash join on each of these batches which are stored in a temporary location in the disk.

When the hash join parameter is disabled, the optimizer chooses the merge join as expected.

## Lessons Learnt:

When hash join is disabled, query optimizer chooses merge join since it performs better when joining large input tables as the cost is the summation of rows in both the tables as opposed to nested loops where the cost is the product of rows in both the tables.

# Experiment 1: Join Algorithms

## Query 2 :

```
SELECT THOUSANDKTUP1.ten, THOUSANDKTUP1.unique2 FROM  
THOUSANDKTUP1, THOUSANDKTUP2 WHERE  
(THOUSANDKTUP1.unique2 = THOUSANDKTUP2.unique2) AND  
(THOUSANDKTUP1.ten < 5)
```

**Parameter Tested :** enable\_mergejoin

**Objective :** To evaluate the execution time of the query with merge join enabled and disabled.

**Expected Results :** The query optimizer is expected to choose merge join as the tables are too large. When this parameter is disabled, we expect the optimizer to choose nested loop join.

# Snapshots

## enable\_mergejoin = On

Query Editor	
1	SET enable_mergejoin=On;
2	EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
3	SELECT THOUSANDKTUP1.ten, THOUSANDKTUP1.unique2
4	FROM THOUSANDKTUP1, THOUSANDKTUP2
5	WHERE (THOUSANDKTUP1.unique2 = THOUSANDKTUP2.unique2)
6	AND (THOUSANDKTUP1.ten < 5)
7	

Data Output	
	QUERY PLAN text
1	Merge Join (actual rows=500042 loops=1)
2	Merge Cond: (thousandktup1.unique2 = thousandktup2.unique2)
3	-> Index Scan using thousandktup1_pkey on thousandktup1 (actual rows=500042 lo...
4	Filter: (ten < 5)
5	Rows Removed by Filter: 499958
6	-> Index Only Scan using thousandktup2_pkey on thousandktup2 (actual rows=1000...
7	Heap Fetches: 1000000
8	Planning Time: 0.373 ms
9	Execution Time: 742.656 ms

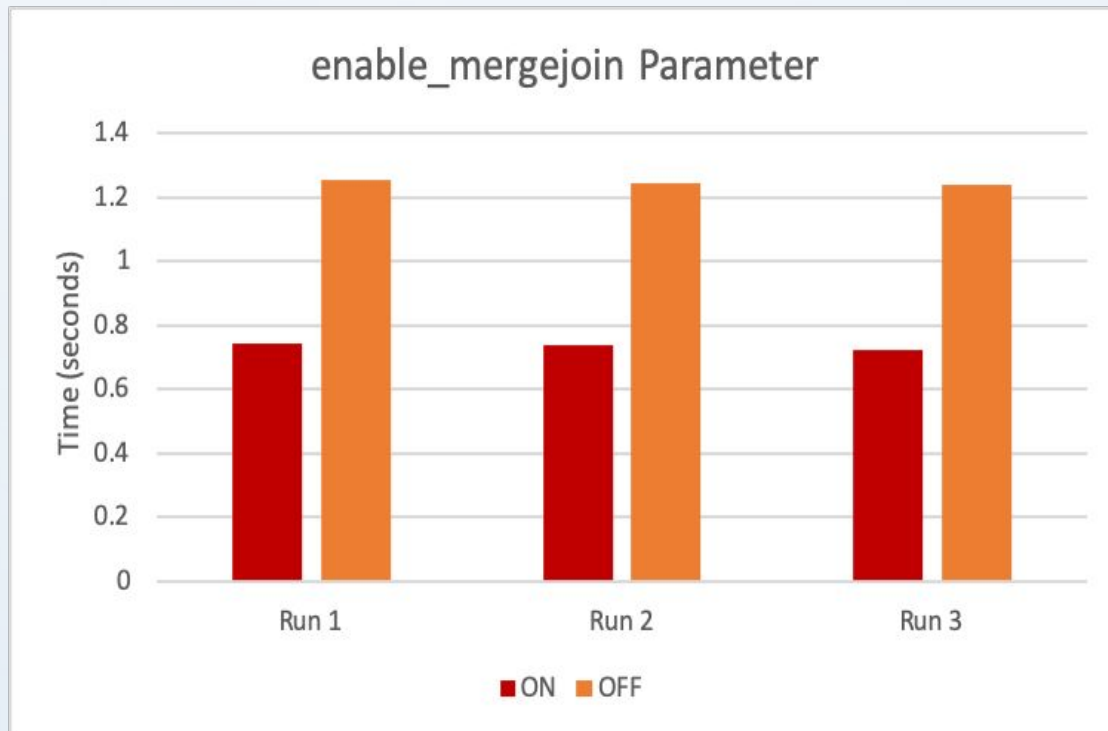
## enable\_mergejoin = off

Query Editor	
1	SET enable_mergejoin=off;
2	EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
3	SELECT THOUSANDKTUP1.ten, THOUSANDKTUP1.unique2
4	FROM THOUSANDKTUP1, THOUSANDKTUP2
5	WHERE (THOUSANDKTUP1.unique2 = THOUSANDKTUP2.unique2)
6	AND (THOUSANDKTUP1.ten < 5)
7	

Data Output	
	QUERY PLAN text
1	Gather (actual rows=500042 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Nested Loop (actual rows=166681 loops=3)
5	-> Parallel Seq Scan on thousandktup1 (actual rows=166681 loops=3)
6	Filter: (ten < 5)
7	Rows Removed by Filter: 166653
8	-> Index Only Scan using thousandktup2_pkey on thousandktup2 (actual rows=1 loo...
9	Index Cond: (unique2 = thousandktup1.unique2)
10	Heap Fetches: 500042
11	Planning Time: 0.184 ms
12	Execution Time: 1280.103 ms



# Experiment 1: Join Algorithms Results



Average Execution time with  
enable\_mergejoin enabled = 0.7351 s

Average Execution time with  
enable\_mergejoin disabled = 1.2469 s

## Results :

When the merge join parameter is enabled, merge join is used to execute this query which is as expected.

When the merge join parameter is disabled, the optimizer chooses the nested loop join which is also as expected.

## Lessons Learnt:

The query optimizer chooses merge join as the tables are large and there exists a clustered index on *unique2* column and the data is already sorted based on this join index. When merge join is disabled, nested loop join is preferred over hash join because of the selectivity rate being more than 10% and the tables being too large to fit into the memory.

## Experiment 2: Memory management and execution time

### Query :

```
SELECT T1.unique2 as T_unique2 FROM THOUSANDKTUP1 T1  
,THOUSANDKTUP2 T2 WHERE T1.unique2=T2.unique2 ORDER BY  
T2.unique3
```

**Parameter Tested :** work\_mem

**Objective :** To evaluate the execution time of the query when work\_mem is varied from 4MB(default) to 128 MB.

**Expected Results :** The execution time is expected to be lesser when the size of the work\_mem is increased to 128 MB from its default size of 4 MB.

# Snapshots

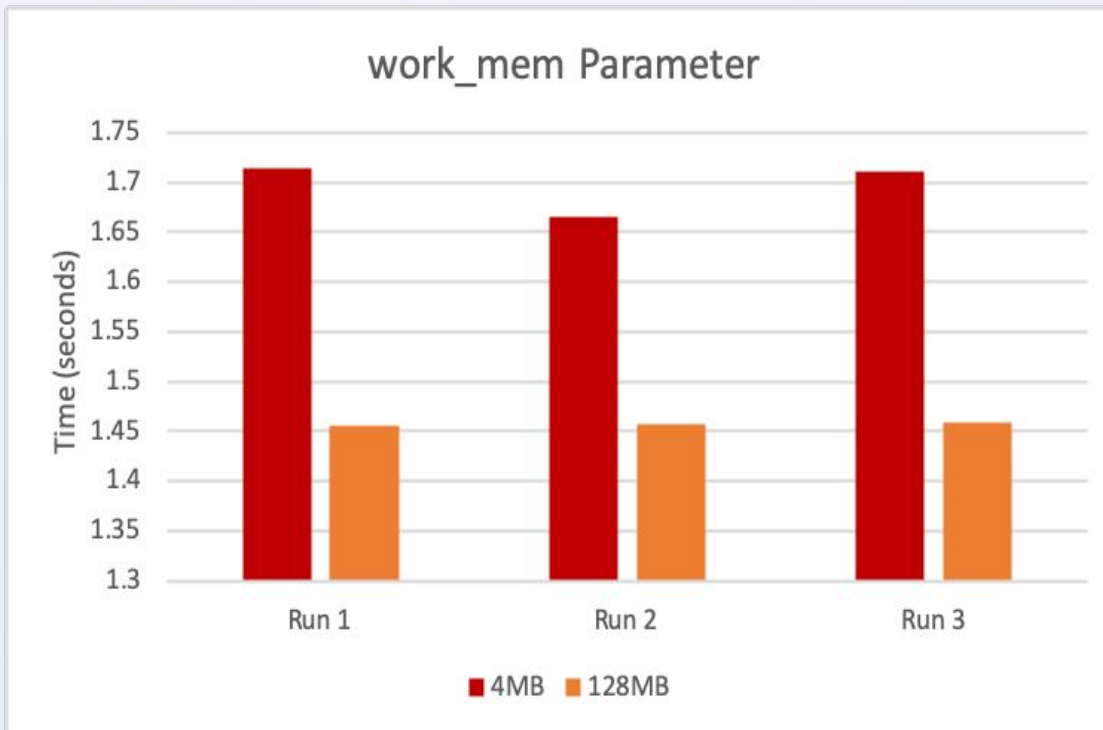
## RESET work\_mem

Query Editor	
1	RESET work_mem;
2	EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
3	SELECT T1.unique2 as T_unique2
4	FROM THOUSANDKTUP1 T1 ,THOUSANDKTUP2 T2
5	WHERE T1.unique2=T2.unique2 ORDER BY T2.unique3
6	
Data Output	
	QUERY PLAN text
1	Gather Merge (actual rows=1000000 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Sort (actual rows=333333 loops=3)
5	Sort Key: t2.unique3
6	Sort Method: external merge Disk: 6464kB
7	Worker 0: Sort Method: external merge Disk: 5304kB
8	Worker 1: Sort Method: external merge Disk: 5960kB
9	-> Parallel Hash Join (actual rows=333333 loops=3)
10	Hash Cond: (t1.unique2 = t2.unique2)
11	-> Parallel Seq Scan on thousandktup1 t1 (actual rows=333333 loops=3)
12	-> Parallel Hash (actual rows=333333 loops=3)
13	Buckets: 131072 Batches: 16 Memory Usage: 3520kB
14	-> Parallel Seq Scan on thousandktup2 t2 (actual rows=333333 loops=...
15	Planning Time: 0.138 ms
16	Execution Time: 1713.450 ms

## SET work\_mem = '128MB'

Query Editor	
1	SET work_mem='128MB';
2	EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
3	SELECT T1.unique2 as T_unique2
4	FROM THOUSANDKTUP1 T1 ,THOUSANDKTUP2 T2
5	WHERE T1.unique2=T2.unique2 ORDER BY T2.unique3
6	
Data Output	
	QUERY PLAN text
1	Sort (actual rows=1000000 loops=1)
2	Sort Key: t2.unique3
3	Sort Method: quicksort Memory: 71452kB
4	-> Hash Join (actual rows=1000000 loops=1)
5	Hash Cond: (t1.unique2 = t2.unique2)
6	-> Seq Scan on thousandktup1 t1 (actual rows=1000000 loops=1)
7	-> Hash (actual rows=1000000 loops=1)
8	Buckets: 1048576 Batches: 1 Memory Usage: 47255kB
9	-> Seq Scan on thousandktup2 t2 (actual rows=1000000 loops=1)
10	Planning Time: 0.137 ms
11	Execution Time: 1492.603 ms

# Experiment 2: Memory Management and execution times Results



Average Execution time when work\_mem is set to 4MB = 1.6969 s

Average Execution time when work\_mem is set to 128 MB = 1.4574 s

## Results :

The execution time of the query when work\_mem parameter is set to 4MB is more than the execution time of the query when the parameter is set to 128 MB. We could note an improved performance as expected when the size of the work\_mem increases. When the size of the work\_mem is 4 MB, the database chooses external merge to sort the data and when the size increases to 128 MB, it chooses quicksort to sort the data.

## Lessons Learnt:

Though the selectivity is more than 10%, the optimizer uses parallel hash join with 16 batches to execute this query. When the memory size is increased there is enough space to fit more data, hence the number of batches has reduced.

# Experiment 3: Aggregations

## Query :

```
SELECT COUNT (oddOnePercent) FROM FIVEMILLIONTUP  
GROUP BY unique1 HAVING unique1>4700676
```

**Parameter Tested :** enable\_hashagg

**Objective :** To evaluate the execution time of the query with hash aggregate enabled and disabled by setting the size of work\_mem to 20 MB.

**Expected Results :** The query optimizer is expected to choose hash aggregate as the selectivity is less than 10%. When this parameter is disabled, we expect the optimizer to choose group aggregation with increased execution time than hash aggregation.



# Snapshots

## enable\_hashagg=On

### Query Editor

```
1 SET enable_hashagg=On;
2 SET work_mem='20MB';
3 EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
4 SELECT COUNT (oddOnePercent) FROM FIVEMILLIONTUP
5 GROUP BY unique1 HAVING unique1>4700676
6
```

### Data Output

	QUERY PLAN	
	text	
1	HashAggregate (actual rows=299323 loops=1)	
2	Group Key: unique1	
3	-> Bitmap Heap Scan on fivemilliontup (actual rows=299323 loops=1)	
4	Recheck Cond: (unique1 > 4700676)	
5	Heap Blocks: exact=131714	
6	-> Bitmap Index Scan on fivemilliontup_unique1_key (actual rows=299323 lo...	
7	Index Cond: (unique1 > 4700676)	
8	Planning Time: 0.158 ms	
9	Execution Time: 1109.721 ms	

## enable\_hashagg=Off

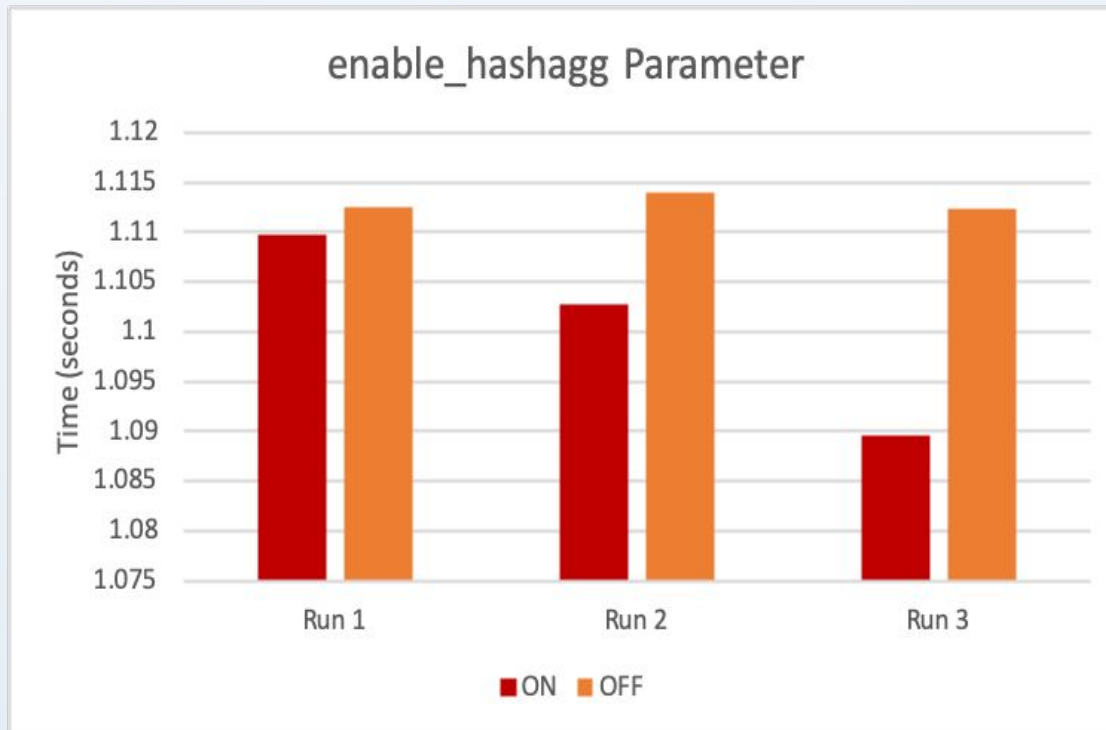
### Query Editor

```
1 SET enable_hashagg=Off;
2 SET work_mem='20MB';
3 EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
4 SELECT COUNT (oddOnePercent) FROM FIVEMILLIONTUP
5 GROUP BY unique1 HAVING unique1>4700676
6
```

### Data Output

	QUERY PLAN	
	text	
1	GroupAggregate (actual rows=299323 loops=1)	
2	Group Key: unique1	
3	-> Sort (actual rows=299323 loops=1)	
4	Sort Key: unique1	
5	Sort Method: external merge Disk: 5280kB	
6	-> Bitmap Heap Scan on fivemilliontup (actual rows=299323 loops=1)	
7	Recheck Cond: (unique1 > 4700676)	
8	Heap Blocks: exact=131714	
9	-> Bitmap Index Scan on fivemilliontup_unique1_key (actual rows=29932...	
10	Index Cond: (unique1 > 4700676)	
11	Planning Time: 0.184 ms	
12	Execution Time: 1113.908 ms	

# Experiment 3: Aggregations Results



Average Execution time with  
enable\_hashagg enabled = 1.1007 s

Average Execution time with  
enable\_hashagg disabled = 1.1124 s

## Results :

As expected, the execution time of the query with hash aggregate is lesser than the execution time of group aggregate (when hashagg is disabled).

## Lessons Learnt:

Here the selectivity is lesser than 10% and the size of work\_mem is 20 MB which is more than the default value, there is sufficient space to fit the hash table, thus the optimizer chooses the hash aggregation. The optimizer would be forced to use group aggregate only when the hash aggregate is disabled and in this case the execution time increases when compared to hash aggregate as it performs an additional job of sorting all the input rows before grouping them.

# Experiment 4: Selectivity

Selectivity less than 10%

## Queries :

```
INSERT INTO TMP
SELECT * FROM FIVEMILLIONTUP
WHERE unique2 < 500000 AND stringu1
LIKE 'AA%'
```

```
INSERT INTO TMP
SELECT * FROM FIVEMILLIONTUP
WHERE unique1 < 500000 AND stringu1
LIKE 'AA%'
```

**Parameter Tested :** enable\_indexscan

**Objective :** To evaluate the execution time of queries using two comparisons, index vs without index (disabling the index scan) and clustered index vs non-clustered index.

**Expected Results :** We expect the optimizer to result with less execution time for queries with index than without index. Among the comparison between clustered and non-clustered index we expect the query with clustered index to do the best in terms of execution time.

# Snapshots

## Clustered index

Query Editor	
1	SET enable_indexscan=On;
2	EXPLAIN (ANALYZE, COSTS OFF,TIMING OFF)
3	INSERT INTO TMP
4	SELECT * FROM FIVEMILLIONTUP
5	WHERE unique2 < 500000 AND string1 LIKE 'AA%'
6	
Data Output	
	<div>QUERY PLAN</div> <div>text</div> <div></div>
1	Insert on tmp (actual rows=0 loops=1)
2	-> Index Scan using fivemilliontup_pkey on fivemilliontup (actual rows=500000 loops=1)
3	Index Cond: (unique2 < 500000)
4	Filter: (string1 ~~ 'AA%':text)
5	Planning Time: 0.199 ms
6	Execution Time: 2544.890 ms

Query Editor	
1	SET enable_indexscan=On;
2	EXPLAIN (ANALYZE, COSTS OFF,TIMING OFF)
3	INSERT INTO TMP
4	SELECT * FROM FIVEMILLIONTUP
5	WHERE unique1 < 500000 AND string1 LIKE 'AA%'
6	
Data Output	
	<div>QUERY PLAN</div> <div>text</div> <div></div>
1	Insert on tmp (actual rows=0 loops=1)
2	-> Bitmap Heap Scan on fivemilliontup (actual rows=500000 loops=1)
3	Recheck Cond: (unique1 < 500000)
4	Rows Removed by Index Recheck: 2907244
5	Filter: (string1 ~~ 'AA%':text)
6	Heap Blocks: exact=47614 lossy=99203
7	-> Bitmap Index Scan on fivemilliontup_unique1_key (actual rows=500000 loops=1)
8	Index Cond: (unique1 < 500000)
9	Planning Time: 0.135 ms
10	Execution Time: 3402.358 ms

## No index

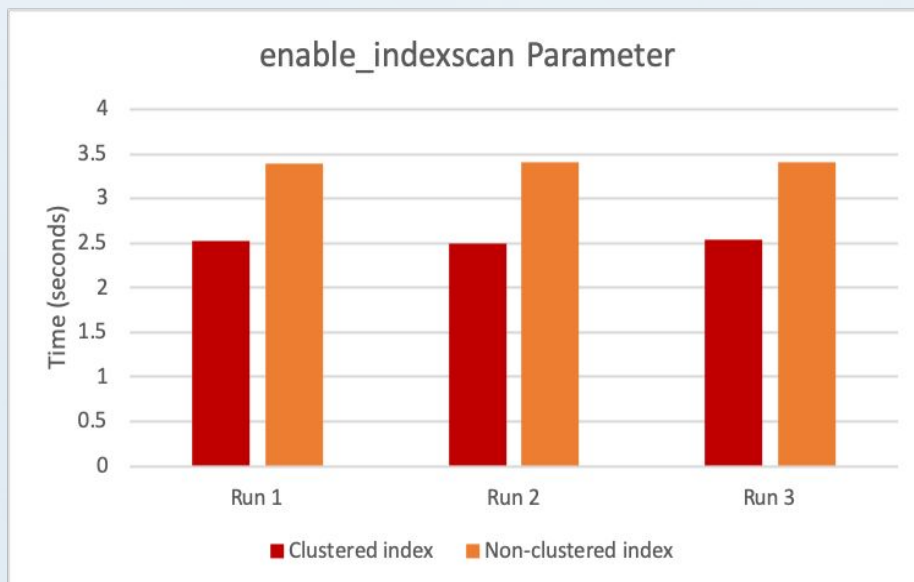
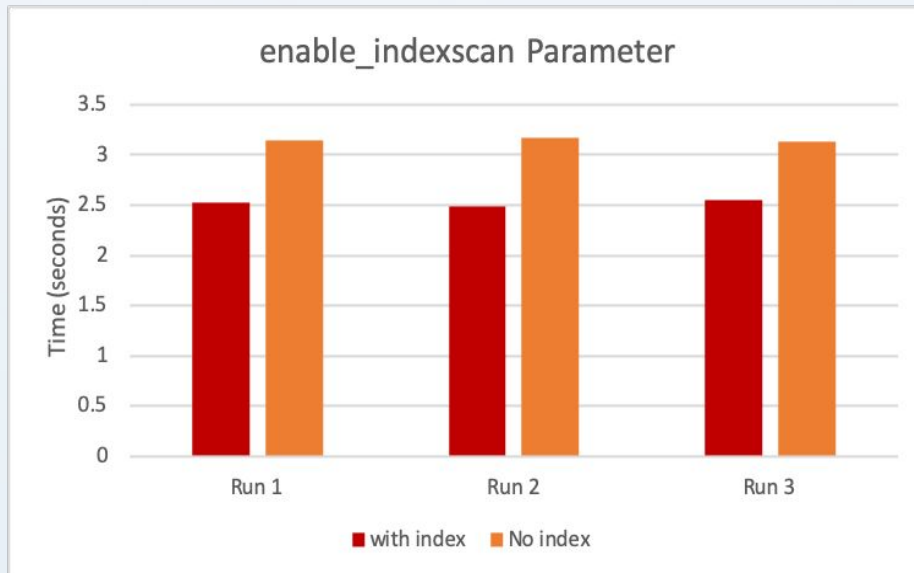
Query Editor	
1	SET enable_indexscan=Off;
2	EXPLAIN (ANALYZE, COSTS OFF,TIMING OFF)
3	INSERT INTO TMP
4	SELECT * FROM FIVEMILLIONTUP
5	WHERE unique2 < 500000 AND string1 LIKE 'AA%'
Data Output	
	<div>QUERY PLAN</div> <div>text</div> <div></div>
1	Insert on tmp (actual rows=0 loops=1)
2	-> Bitmap Heap Scan on fivemilliontup (actual rows=500000 loops=1)
3	Recheck Cond: (unique2 < 500000)
4	Filter: (string1 ~~ 'AA%':text)
5	Heap Blocks: exact=15152
6	-> Bitmap Index Scan on fivemilliontup_pkey (actual rows=500000 loops=1)
7	Index Cond: (unique2 < 500000)
8	Planning Time: 0.117 ms
9	Execution Time: 3174.598 ms

## Non-Clustered index



# Experiment 4: Selectivity Results

Selectivity less than 10%



## Results :

As expected, the execution time of the query with clustered index (enable\_indexscan=ON) is lesser than the execution time of the query with no index (enable\_indexscan=OFF) and non-clustered index. When index scan is disabled the query uses bitmap index scan which is also a form of index scan. In case of non-clustered index query, optimizer chooses bitmap scan though index scan parameter is enabled this is because of non-clustered index (tuples not sorted) set on *unique1* column that takes time to scan through all tuples.

## Lessons Learnt:

Index scan is most efficient when selectivity is lesser than 10%. When index scan is disabled, Bitmap index scan is chosen as the next best scan. It is opted when select tuples are too less for sequential scan but too large for index scan.

Average Execution time with No index= 3.1514 s

Average Execution time with Clustered index = 2.5186 s

Average Execution time with Non-Clustered index = 3.4011 s



# Experiment 4: Selectivity

Selectivity more than 10%

## Query :

```
SELECT * FROM FIVEMILLIONTUP  
WHERE unique2 > 97200 AND stringu1 LIKE 'AA%'
```

**Parameter Tested :** enable\_seqscan

**Objective :** To evaluate the execution time of the query with more than 10% selectivity, by toggling the sequential scan parameter.

**Expected Results :** The query optimizer is expected to choose sequential scan as the selectivity is more than 10%. When this parameter is disabled, we expect the optimizer to choose index scan with increased execution time than sequential scan.

# Snapshots

## enable\_seqscan=On

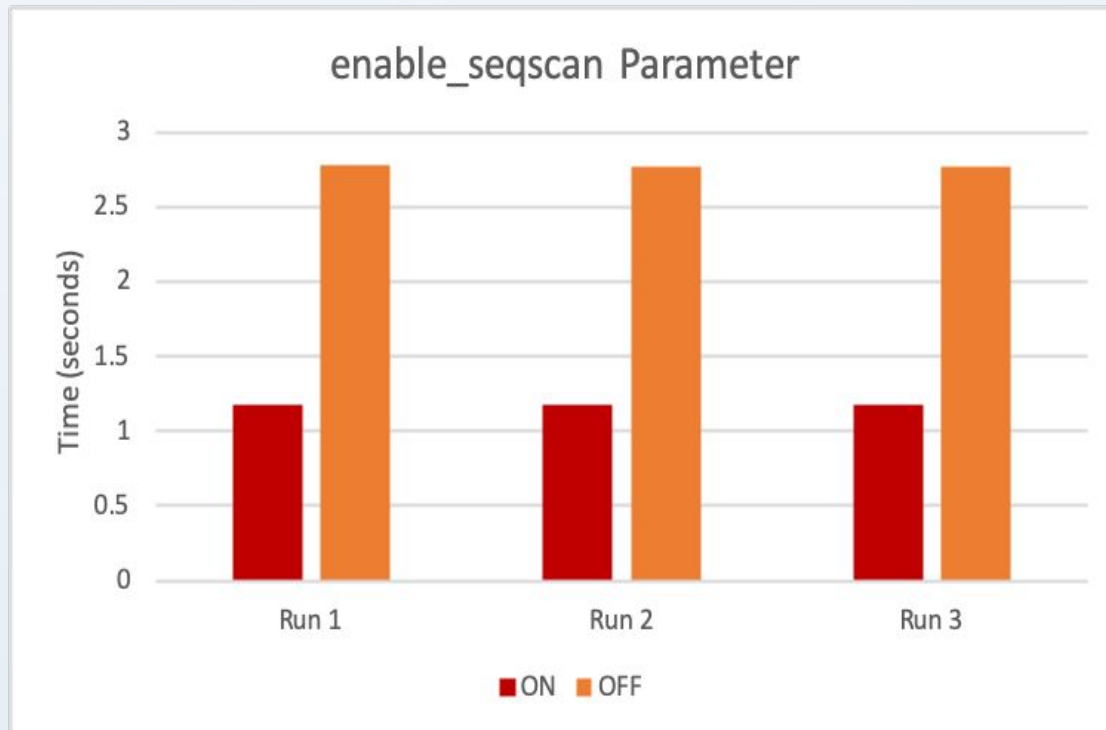
Query Editor	
1	SET enable_seqscan=On;
2	SET enable_indexscan=On;
3	EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
4	SELECT * FROM FIVEMILLIONTUP
5	WHERE unique2 > 97200 AND stringu1 LIKE 'AA%'
6	
Data Output	
	<div>QUERY PLAN</div> <div>text</div> <div></div>
1	Seq Scan on fivemilliontup (actual rows=4902799 loops=1)
2	Filter: ((unique2 > 97200) AND (stringu1 ~~ 'AA%':text))
3	Rows Removed by Filter: 97201
4	Planning Time: 0.104 ms
5	Execution Time: 1177.784 ms

## enable\_seqscan=Off

Query Editor	
1	SET enable_seqscan=Off;
2	SET enable_indexscan=On;
3	EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
4	SELECT * FROM FIVEMILLIONTUP
5	WHERE unique2 > 97200 AND stringu1 LIKE 'AA%'
6	
Data Output	
	<div>QUERY PLAN</div> <div>text</div> <div></div>
1	Index Scan using fivemilliontup_pkey on fivemilliontup (actual rows=4902799 lo...
2	Index Cond: (unique2 > 97200)
3	Filter: (stringu1 ~~ 'AA%':text)
4	Planning Time: 0.157 ms
5	Execution Time: 2777.315 ms

# Experiment 4: Selectivity Results

Selectivity more than 10%



Average Execution time with  
enable\_seqscan enabled = 1.1826 s

Average Execution time with  
enable\_seqscan disabled = 2.7715 s

## Results :

Though a clustered index exists on *unique2*, the query optimizer only opts for sequential scan to execute the query, as expected this is because the selectivity is more than 10% (when enable\_seqscan is on). Also the optimizer as predicted was forced to opt for index scan when the sequential scan parameter is disabled resulting with more execution time.

## Lessons Learnt:

Here the selectivity is more than 10% so sequential scan is best suited since index scan performs worse. But when the sequential scan is disabled optimizer prioritizes index scan over bitmap scan and executes the query with index scan. It is impossible to suppress sequential scans entirely, but turning this variable off discourages the optimizer from using one if there are other methods available.

# Summary

- Most preferred join is the *hash join* it is chosen when one of the two relations is relatively smaller than the other and when the projections of the join tables are not already sorted on join columns. The query optimizer picks *merge join* when the relations are large and the projections of the joined tables are already sorted.
- Large size of *work\_mem* can decrease the execution time of the queries since now there is enough space in the memory to fit the data in. But when the size of *work\_mem* is increased too far, we might lose memory that might be required for read cache and shared buffers and a large *work\_mem* value for a small relational database is not efficient.
- When selectivity is lesser than 10%, the optimizer chooses *hash aggregate* as the hash table fits in memory. This operation does not require a presorted data set, but uses large memory to carry out this process. *Group aggregate* uses a presorted dataset according to group by clause and does not need a buffer to store large amounts of data.
- *Index scan* performs B-tree traversal to find matches and then fetches the corresponding data. It is opted when selectivity is lesser than 10%. *Sequential scan* sequentially scans pages and returns data. This is used when selectivity is more than 10% and is used commonly to extract data from the disk.

# Lessons Learnt

- It was surprising to note Postgres chose hash join to execute a query with large tables. Multi batch hash join divides large tables into batches of work\_mem size and then applies hash join on each batch. This took us aback as we expected merge join to execute instead.
- Execution of a query seemed to improve after a couple of runs as the data was being cached. The shared buffer and cache needs to be refreshed timely to avoid this and to get better analysis of performance.
- Sometimes default values used by the optimizer is not well suited for certain queries. For instance work\_mem value of 4MB is not optimal for a database that executes more sorting and hashing functions.
- We got introduced to Bitmap heap scan, which the optimizer preferred in cases when the tuples obtained had select read that were too large for index scan and too small for sequential scan.



# References

- <https://www.postgresql.org/docs/9.4/runtime-config-query.html>
- <https://www.postgresql.org/docs/9.4/runtime-config-resource.html>
- <https://use-the-index-luke.com/sql/explain-plan/postgresql/operations>
- The Wisconsin Benchmark: Past, Present, and Future - David J. DeWitt
- <https://dba.stackexchange.com/questions/27893/increasing-work-mem-and-shared-buffers-on-postgres-9-2-significantly-slows-down>

Thank You

