# DBMS Implementation project Part III

Pooja Nalan & Utsav Raychaudhuri

# Postgres

It is a powerful, open-source object relational database that extends SQL language.

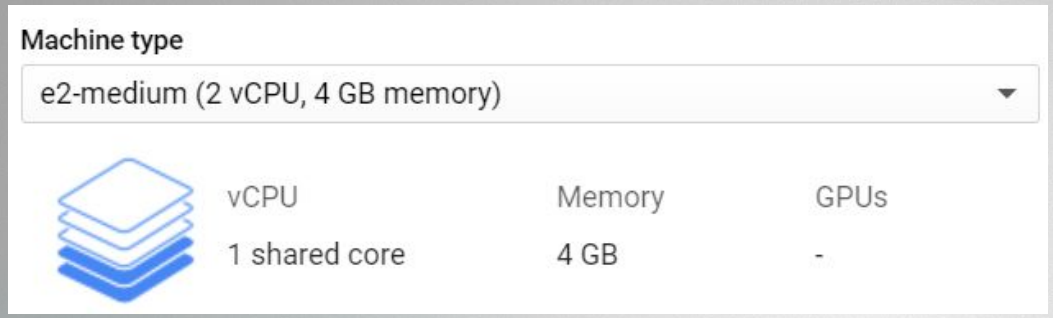
Robust feature set, extensible and consistent delivery of performance and runs on major operating systems.

It has many features that help developers build fault tolerant environment and manage data no matter how big.

# Why Postgres?

- Familiarity with the system and ease of use
- Easier to compare differences in the performance within a system and test various features
- Lots of resources and community threads to learn from.
- Various query parameters can be tweaked and observed for variations in performance.

# System specification of Virtual Machine in Google Cloud Platform





**Database: Postgresql**

```
pnalan@instance-1:/usr/lib/postgresql/11/bin$ psql -V
psql (PostgreSQL) 11.10 (Debian 11.10-0+deb10u1)
```

# Goals

- Using the Wisconsin benchmark design to generate data tables of various sizes and perform various experiments on them. Our code generates data according to the Wisconsin benchmark specification, as described in “The Wisconsin Benchmark: Past, Present, and Future” by David J. DeWitt

- Test the performance of the system by choosing a variety of queries fetching data of different size and attributes under varying conditions.

- Observe and record the results of execution and measure the performance in terms of time and cost and the impact of modifying query parameters.

# Experiment 1

- **Performance issue** :

work_mem specifies the amount of memory used by internal sort operations and hash tables before writing to temporary disk files.There could be a difference  in the performance when the work_mem is increased or decreased based on the no of connections and also the sort operation performed on the relations.

- **Parameters** :

Vary work_mem upto 128 MB.

# Execution

**Query**: EXPLAIN ANALYZE SELECT unique2,string4 FROM hundredktup ORDER BY string4;

```
dbmsproject=# EXPLAIN ANALYZE SELECT unique2,string4 FROM hundredktup ORDER BY string4;
                                                                 QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=19860.82..20110.82 rows=100000 width=57) (actual time=118.527..133.501 rows=100000 loops=1)
   Sort Key: string4
   Sort Method: external merge  Disk: 6576kB
   ->  Seq Scan on hundredktup  (cost=0.00..4031.00 rows=100000 width=57) (actual time=0.619..66.123 rows=100000 loops=1)
 Planning Time: 7.485 ms
 Execution Time: 139.082 ms
(6 rows)
```

**work_mem = ‘500kB’**

```
dbmsproject=# EXPLAIN ANALYZE SELECT unique2,string4 FROM hundredktup ORDER BY string4;
                                                                 QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=16098.32..16348.32 rows=100000 width=57) (actual time=90.876..106.671 rows=100000 loops=1)
   Sort Key: string4
   Sort Method: external merge  Disk: 6584kB
   ->  Seq Scan on hundredktup  (cost=0.00..4031.00 rows=100000 width=57) (actual time=0.034..37.254 rows=100000 loops=1)
 Planning Time: 4.638 ms
 Execution Time: 114.269 ms
(6 rows)
```

**work_mem = ‘4MB’**

```
dbmsproject=# set work_mem = '50MB';
SET
dbmsproject=# EXPLAIN ANALYZE SELECT unique2,string4 FROM hundredktup ORDER BY string4;
                                                                 QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=12335.82..12585.82 rows=100000 width=57) (actual time=71.055..84.447 rows=100000 loops=1)
   Sort Key: string4
   Sort Method: quicksort  Memory: 17135kB
   ->  Seq Scan on hundredktup  (cost=0.00..4031.00 rows=100000 width=57) (actual time=0.029..37.034 rows=100000 loops=1)
 Planning Time: 4.386 ms
 Execution Time: 89.314 ms
(6 rows)
```

**work_mem = ‘50MB’**

# Results of Execution

| Exp No | Execution time for work_mem (milli seconds) | | | |
|---|---|---|---|---|
| | 500kB | 4 MB | 50 MB | 100 MB |
| 1 | 129.75 | 105.241 | 89.314 | 80.605 |
| 2 | 131.942 | 109.163 | 89.793 | 91.961 |
| 3 | 139.082 | 110.789 | 100.925 | 91.966 |
| 4 | 140.871 | 114.269 | 101.712 | 93.065 |
| 5 | 142.714 | 116.323 | 103.514 | 94.113 |
| Average | 137.298 | 111.407 | 97.476 | 92.330 |

# Results and Lessons Learned

The size of the relation hundredktup is 46MB. Since the size of the relation is larger than work_mem which is 4MB by default, the database had to use external merge sort to sort the relation.

Now changing the work_mem to 50 MB and then 100 MB decreases the execution, the database uses quicksort which reduces the execution time from 111.407 to 92.330

Increasing work_mem doesn't always improve the performance. A database backs a web server, and, in that case, you have many queries coming through the web service and too many hitting at the same time. To not overwhelm the memory, we want those different queries to get a small bit of memory. We want work mem to be relatively small to allow many queries and to allocate some amount of space to each query.

# Experiment 2

- **Performance Issue**: This experiment explores the benefit of setting the random_page_cost parameter on the performance of queries.
- **What is random_page_cost and what does it do?**

Sets the planner's estimate of the cost of a non-sequentially-fetched disk page. The default is 4.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name.Reducing this value relative to seq_page_cost will cause the system to prefer index scans; raising it will make index scans look relatively more expensive. We can raise or lower both values together to change the importance of disk I/O costs relative to CPU costs

# The Experiment

```
dbmsproject=# set random_page_cost=1000000000;
SET
dbmsproject=# select * from hundredktup where unique2=9999;
dbmsproject=# explain analyze select * from hundredktup where unique2=9999;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on hundredktup  (cost=0.00..4281.00 rows=1 width=211) (actual time=1.616..15.914 rows=1 loops=1)
   Filter: (unique2 = 9999)
   Rows Removed by Filter: 99999
 Planning Time: 0.216 ms
 Execution Time: 16.002 ms
(5 rows)

dbmsproject=# set random_page_cost=1;
SET
dbmsproject=# explain analyze select * from hundredktup where unique2=9999;
                                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Index Scan using hundredktup_pkey on hundredktup  (cost=0.29..2.31 rows=1 width=211) (actual time=1.563..1.566 rows=1 loops=1)
   Index Cond: (unique2 = 9999)
 Planning Time: 0.272 ms
 Execution Time: 1.593 ms
(4 rows)

dbmsproject=#
```

# Result of Execution

```
postgres=# \c dbmsproject
You are now connected to database "dbmsproject" as user "postgres".
dbmsproject=# set random_page_cost=1;
SET
dbmsproject=# explain analyse select * from onemtup where unique2=155835;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Index Scan using onemtup_pkey1 on onemtup  (cost=0.42..2.44 rows=1 width=211) (actual time=3.520..3.525 rows=1 loops=1)
   Index Cond: (unique2 = 155835)
 Planning Time: 22.596 ms
 Execution Time: 3.689 ms
(4 rows)
```

```
postgres=# \c dbmsproject
You are now connected to database "dbmsproject" as user "postgres".
dbmsproject=# set random_page_cost=100;
SET
dbmsproject=# explain analyse select * from onemtup where unique2=155835;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Index Scan using onemtup_pkey1 on onemtup  (cost=0.42..200.44 rows=1 width=211) (actual time=3.352..3.358 rows=1 loops=1)
   Index Cond: (unique2 = 155835)
 Planning Time: 18.527 ms
 Execution Time: 3.460 ms
(4 rows)
```

```
dbmsproject=# set random_page_cost=1000;
SET
dbmsproject=# explain analyse select * from onemtup where unique2=155835;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Index Scan using onemtup_pkey1 on onemtup  (cost=0.42..2000.44 rows=1 width=211) (actual time=2.856..2.859 rows=1 loops=1)
   Index Cond: (unique2 = 155835)
 Planning Time: 16.950 ms
 Execution Time: 2.993 ms
(4 rows)

dbmsproject=# \q
```

```
postgres=# set random_page_cost=10000;
SET
postgres=# explain analyse select * from onemtup where unique2=155835;
ERROR:  relation "onemtup" does not exist
LINE 1: explain analyse select * from onemtup where unique2=155835;
                                      ^
postgres=# \c dbmsproject
You are now connected to database "dbmsproject" as user "postgres".
dbmsproject=# explain analyse select * from onemtup where unique2=155835;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Index Scan using onemtup_pkey1 on onemtup  (cost=0.42..8.44 rows=1 width=211) (actual time=3.262..3.267 rows=1 loops=1)
   Index Cond: (unique2 = 155835)
 Planning Time: 17.806 ms
 Execution Time: 3.399 ms
(4 rows)
```

# Result of Execution(Continued)

```
postgres=# set random_page_cost=100000;
SET
postgres=# explain analyse select * from onemtup where unique2=155835;
ERROR:  relation "onemtup" does not exist
LINE 1: explain analyse select * from onemtup where unique2=155835;
                                     ^
postgres=# \c dbmsproject
You are now connected to database "dbmsproject" as user "postgres".
dbmsproject=# explain analyse select * from onemtup where unique2=155835;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Index Scan using onemtup_pkey1 on onemtup  (cost=0.42..8.44 rows=1 width=211) (actual time=4.190..4.195 rows=1 loops=1)
   Index Cond: (unique2 = 155835)
 Planning Time: 17.483 ms
 Execution Time: 4.298 ms
(4 rows)
```

```
postgres=# \c dbmsproject
You are now connected to database "dbmsproject" as user "postgres".
dbmsproject=# set random_page_cost=10000000;
SET
dbmsproject=# explain analyse select * from onemtup where unique2=155835;
                                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..36512.43 rows=1 width=211) (actual time=1554.074..1558.604 rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on onemtup  (cost=0.00..35512.33 rows=1 width=211) (actual time=1079.317..1547.148 rows=0 loops=3)
         Filter: (unique2 = 155835)
         Rows Removed by Filter: 333333
 Planning Time: 19.578 ms
 Execution Time: 1558.720 ms
(8 rows)
```

# Results and Lessons Learned

| Exp No. | random_page_cost | Execution time | Type of Scan Used |
|---|---|---|---|
| 1 | 1 | 3.689 | Index |
| 2 | 100 | 3.4 | Index |
| 3 | 10000 | 3.4 | Index |
| 4 | 100000 | 4.3 | Index |
| 5 | 10000000 | 1558.72 | Sequential |

- We saw 16 times improvement in performance by reducing the value of random_page_cost. So, what this parameter does is by setting this value higher the query optimizer thinks that a sequential scan is better than an index scan which was the case in our experiment.
- So, it would be good to set this to a lower value because by doing this the query optimizer prefers an index scan which took only 1 ms in our case to scan a 100000-tuple dataset.

# Experiment 3

- **Performance issue:**
- This test explores how the query optimizer benefits from using a clustered index when compared to non-clustered index.
- **Parameters:**
- The parameter we would be using here is unique2 which is a clustered index and unique1 which is a non-clustered index.

**Query**
**Example 1:**

Clustered Index:

```
EXPLAIN ANALYZE SELECT * FROM tenktup1, (SELECT * FROM tenktup2 WHERE tenktup2.unique2 < 1000) AS BPRIME WHERE (tenktup1.unique2 = BPRIME.unique2);
```

Non-Clustered Index:

```
EXPLAIN ANALYZE SELECT * FROM tenktup1, (SELECT * FROM tenktup2 WHERE tenktup2.unique1 < 1000) AS BPRIME WHERE (tenktup1.unique1 = BPRIME.unique1);
```

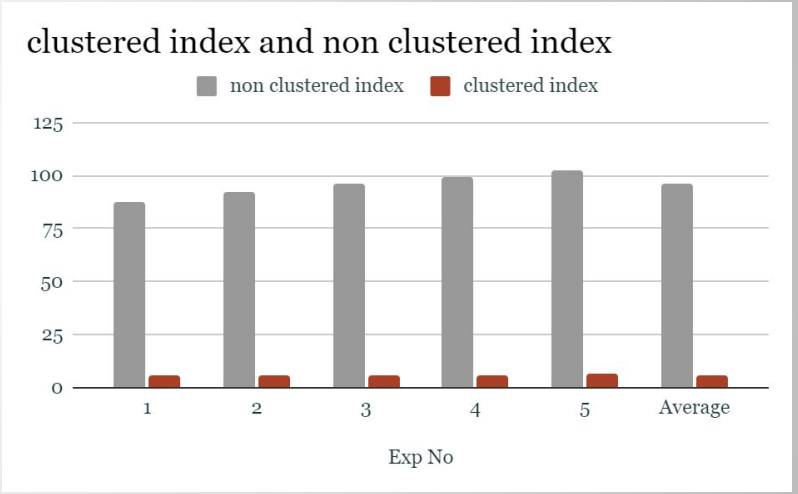
# Execution Example 1

```
dbmsproject=# EXPLAIN ANALYZE SELECT * FROM tenktup1, (SELECT * FROM tenktup2 WHERE
tenktup2.unique2 < 1000) AS BPRIME WHERE (tenktup1.unique2 = BPRIME.unique2);
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=77.28..507.55 rows=1000 width=422) (actual time=0.591..5.742 rows=1000 loops=1)
   Hash Cond: (tenktup1.unique2 = tenktup2.unique2)
   ->  Seq Scan on tenktup1  (cost=0.00..404.00 rows=10000 width=211) (actual time=0.007..2.172 rows=10000 loops=1)
   ->  Hash  (cost=64.78..64.78 rows=1000 width=211) (actual time=0.574..0.575 rows=1000 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 246kB
         ->  Index Scan using tenktup2_pkey on tenktup2  (cost=0.29..64.78 rows=1000 width=211) (actual time=0.016..0.312 rows=1000 loops=1)
               Index Cond: (unique2 < 1000)
 Planning Time: 0.455 ms
 Execution Time: 5.889 ms
(9 rows)
```

```
dbmsproject=# EXPLAIN ANALYZE SELECT * FROM tenktup1, (SELECT * FROM tenktup2 WHERE
tenktup2.unique1 < 1000) AS BPRIME WHERE (tenktup1.unique1 = BPRIME.unique1);
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=361.04..791.30 rows=1000 width=422) (actual time=77.192..96.138 rows=1000 loops=1)
   Hash Cond: (tenktup1.unique1 = tenktup2.unique1)
   ->  Seq Scan on tenktup1  (cost=0.00..404.00 rows=10000 width=211) (actual time=0.979..15.953 rows=10000 loops=1)
   ->  Hash  (cost=348.54..348.54 rows=1000 width=211) (actual time=76.137..76.140 rows=1000 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 246kB
         ->  Bitmap Heap Scan on tenktup2  (cost=24.04..348.54 rows=1000 width=211) (actual time=2.258..74.621 rows=1000 loops=1)
               Recheck Cond: (unique1 < 1000)
               Heap Blocks: exact=294
               ->  Bitmap Index Scan on tenktup2_unique1_key  (cost=0.00..23.79 rows=1000 width=0) (actual time=1.571..1.571 rows=1000 loops=1)
                     Index Cond: (unique1 < 1000)
 Planning Time: 24.484 ms
 Execution Time: 96.418 ms
(12 rows)
```

# Result of Execution:

| Exp No | Execution time (milli seconds) | |
|---|---|---|
| | non clustered index | clustered index |
| 1 | 87.118 | 5.385 |
| 2 | 92.079 | 5.431 |
| 3 | 96.418 | 5.889 |
| 4 | 99.367 | 5.892 |
| 5 | 102.351 | 6.395 |
| **Average** | **95.95466667** | **5.737333333** |



**Query Example 2:**

Clustered Index:

```
EXPLAIN ANALYZE SELECT * FROM onektup LEFT JOIN (tenktup1 JOIN tenktup2 ON (tenktup1.unique2 = tenktup2.unique2)) ON (onektup.unique2 = tenktup1.unique2) AND tenktup1.tenpercent = 9;
```

Non - Clustered Index:

```
EXPLAIN ANALYZE SELECT * FROM onektup LEFT JOIN (tenktup1 JOIN tenktup2 ON (tenktup1.unique1 = tenktup2.unique1)) ON (onektup.unique1 = tenktup1.unique1) AND tenktup1.tenpercent = 9;
```

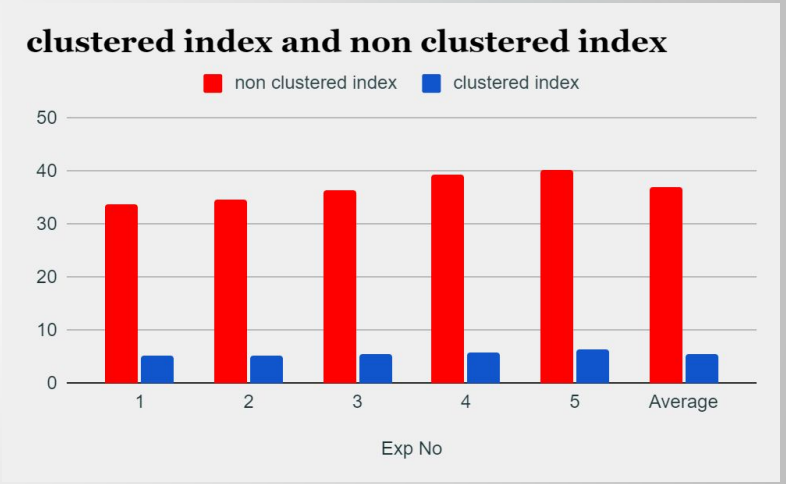
# Execution Example 2:

```
dbmsproject=# EXPLAIN ANALYZE SELECT * FROM onektup LEFT JOIN (tenktup1 JOIN tenktup2 ON (tenktup1.unique1 = tenktup2.unique1)) ON (onektup.unique1 = tenktup1.uni
que1) AND tenktup1.tenpercent = 9;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Hash Right Join  (cost=495.00..935.89 rows=1000 width=633) (actual time=18.889..36.026 rows=1000 loops=1)
   Hash Cond: (tenktup1.unique1 = onektup.unique1)
   ->  Hash Join  (cost=441.50..879.76 rows=1000 width=422) (actual time=15.266..31.189 rows=1000 loops=1)
         Hash Cond: (tenktup2.unique1 = tenktup1.unique1)
         ->  Seq Scan on tenktup2  (cost=0.00..412.00 rows=10000 width=211) (actual time=1.228..13.713 rows=10000 loops=1)
         ->  Hash  (cost=429.00..429.00 rows=1000 width=211) (actual time=13.999..14.000 rows=1000 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 246kB
               ->  Seq Scan on tenktup1  (cost=0.00..429.00 rows=1000 width=211) (actual time=1.142..13.469 rows=1000 loops=1)
                     Filter: (tenpercent = 9)
                     Rows Removed by Filter: 9000
   ->  Hash  (cost=41.00..41.00 rows=1000 width=211) (actual time=3.573..3.574 rows=1000 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 246kB
         ->  Seq Scan on onektup  (cost=0.00..41.00 rows=1000 width=211) (actual time=0.730..3.212 rows=1000 loops=1)
 Planning Time: 33.218 ms
 Execution Time: 36.356 ms
(15 rows)
```

```
dbmsproject=# EXPLAIN ANALYZE SELECT * FROM onektup LEFT JOIN (tenktup1 JOIN tenktup2 ON (tenktup1.unique2 = tenktup2.unique2)) ON (onektup.unique2 = tenktup1.uni
que2) AND tenktup1.tenpercent = 9;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Merge Right Join  (cost=479.39..614.64 rows=1000 width=633) (actual time=3.208..5.337 rows=1000 loops=1)
   Merge Cond: (tenktup1.unique2 = onektup.unique2)
   ->  Merge Join  (cost=479.11..1104.11 rows=1000 width=422) (actual time=3.187..3.892 rows=98 loops=1)
         Merge Cond: (tenktup2.unique2 = tenktup1.unique2)
         ->  Index Scan using tenktup2_pkey on tenktup2  (cost=0.29..585.28 rows=10000 width=211) (actual time=0.014..0.456 rows=1016 loops=1)
         ->  Sort  (cost=478.83..481.33 rows=1000 width=211) (actual time=3.157..3.169 rows=98 loops=1)
               Sort Key: tenktup1.unique2
               Sort Method: quicksort  Memory: 290kB
               ->  Seq Scan on tenktup1  (cost=0.00..429.00 rows=1000 width=211) (actual time=0.009..2.784 rows=1000 loops=1)
                     Filter: (tenpercent = 9)
                     Rows Removed by Filter: 9000
   ->  Index Scan using onektup_pkey on onektup  (cost=0.28..69.28 rows=1000 width=211) (actual time=0.014..0.318 rows=1000 loops=1)
 Planning Time: 5.247 ms
 Execution Time: 5.739 ms
(14 rows)
```

# Results of Execution

| Exp No | Execution time (milli seconds ) | |
|---|---|---|
| | non clustered index | clustered index |
| 1 | 33.862 | 5.221 |
| 2 | 34.76 | 5.27 |
| 3 | 36.356 | 5.526 |
| 4 | 39.443 | 5.739 |
| 5 | 40.331 | 6.395 |
| Average | **36.853** | **5.511666667** |



# Results and Lessons Learned

In clustered index the tuples in the table are sorted on the index and stored in memory. By default, the primary key of a table has a clustered index. Once an index search is done and found, pulling out the data on the same page is faster since we've already found the start point and it's easier to point all successive data nearby.

Since, each record is found from index data, it needs to look in heap data which causes random I/O and it's considered to perform slower than sequential I/O . If a small percentage of records are selected which is calculated based on the **PostgreSQL** optimizer cost, only then PostgreSQL chooses Index scan otherwise it continues to use sequence scan even though there is an index on the table.

We expect to see a higher execution time when we query a non clustered index when compared to using a clustered index. Even though index creation speeds up performance , it has to be chosen carefully as it has overhead in terms of storage and has many factors such as database structure, queries , etc.

# Experiment 4

- **Performance Issue**:

  We analyze the impact of various join algorithms on the query execution time.

- **Data set**:

  The dataset we use here is a join on tenktup and onemtup which consists of ten thousand and one million records respectively.

- **Query:**

  In this query, we extract all the records of onemtup and tenktup on which a join operation is performed. All the records where tenktup's unique2 values are lesser than 7000 are fetched and analyzed.

  EXPLAIN ANALYZE SELECT * FROM tenktup1,onemtup WHERE onemtup.unique2 = tenktup1.unique2 AND onemtup.unique2<7000

  We expect this to use a sort merge join and this should be faster because we already have a clustered index on unique2

  SELECT * FROM hundredktup h1,hundredktup h2 WHERE h1.unique1 = h2.unique1 AND h1.unique1 BETWEEN 7000 AND 10000;

  We expect this to use a hash join which should take more time than the sort merge join. It was very interesting to see this same query when executed on onemtup the query optimizer resorts to a nested loop join. The reason is still a little unclear to us.

# Execution

## Sort Merge Join

```
dbmsproject=# EXPLAIN ANALYZE SELECT * FROM tenktup1,onemtup WHERE onemtup.unique2 = tenktup1.unique2 AND onemtup.unique2 between 7000 and 10000;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------
 Merge Join  (cost=0.83..605.31 rows=32 width=422) (actual time=24.203..37.581 rows=3000 loops=1)
   Merge Cond: (tenktup1.unique2 = onemtup.unique2)
   ->  Index Scan using tenktup1_pkey on tenktup1  (cost=0.29..577.28 rows=10000 width=211) (actual time=0.008..19.225 rows=10000 loops=1)
   ->  Index Scan using onemtup_pkey1 on onemtup  (cost=0.42..201.85 rows=3221 width=211) (actual time=6.051..16.012 rows=3000 loops=1)
         Index Cond: ((unique2 >= 7000) AND (unique2 <= 10000))
 Planning Time: 85.076 ms
 Execution Time: 37.898 ms
(7 rows)
```

## Hash Join

```
dbmsproject=# EXPLAIN ANALYZE SELECT * FROM tenktup1,tenktup2 WHERE tenktup2.unique1 = tenktup1.unique1 AND tenktup1.unique1 BETWEEN 7000 AND 10000;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=413.77..852.03 rows=3000 width=422) (actual time=103.049..116.396 rows=3000 loops=1)
   Hash Cond: (tenktup2.unique1 = tenktup1.unique1)
   ->  Seq Scan on tenktup2  (cost=0.00..412.00 rows=10000 width=211) (actual time=0.483..10.230 rows=10000 loops=1)
   ->  Hash  (cost=376.27..376.27 rows=3000 width=211) (actual time=102.501..102.502 rows=3000 loops=1)
         Buckets: 4096  Batches: 1  Memory Usage: 744kB
         ->  Index Scan using tenktup1_unique1_key on tenktup1  (cost=0.29..376.27 rows=3000 width=211) (actual time=0.906..100.908 rows=3000 loops=1)
               Index Cond: ((unique1 >= 7000) AND (unique1 <= 10000))
 Planning Time: 18.877 ms
 Execution Time: 116.685 ms
(9 rows)
```

# Execution

## Nested Loop Join

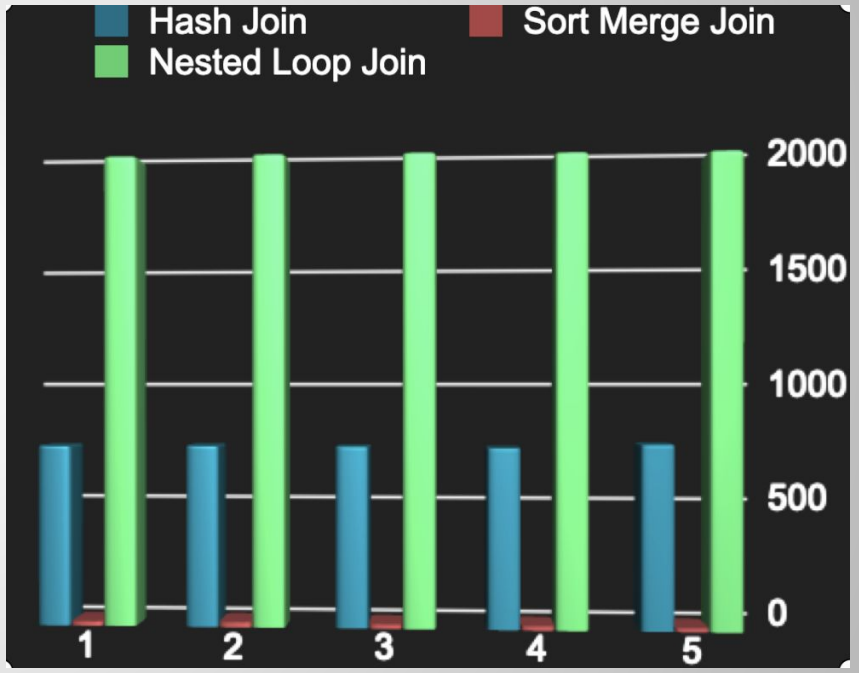
```
dbmsproject=# EXPLAIN ANALYZE SELECT * FROM onemtup o1,onemtup o2 WHERE o1.unique1 = o2.unique1 AND o1.unique1 BETWEEN 7000 AND 10000;
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1075.16..22446.08 rows=2957 width=422) (actual time=1.580..17.929 rows=3001 loops=1)
   Workers Planned: 1
   Workers Launched: 1
   ->  Nested Loop  (cost=75.16..21150.38 rows=1739 width=422) (actual time=0.639..10.115 rows=1500 loops=2)
         ->  Parallel Bitmap Heap Scan on onemtup o1  (cost=74.73..8800.08 rows=1739 width=211) (actual time=0.619..4.539 rows=1500 loops=2)
               Recheck Cond: ((unique1 >= 7000) AND (unique1 <= 10000))
               Heap Blocks: exact=2065
               ->  Bitmap Index Scan on onemtup_unique1_key1  (cost=0.00..73.99 rows=2957 width=0) (actual time=0.740..0.741 rows=3001 loops=1)
                     Index Cond: ((unique1 >= 7000) AND (unique1 <= 10000))
         ->  Index Scan using onemtup_unique1_key1 on onemtup o2  (cost=0.42..7.10 rows=1 width=211) (actual time=0.003..0.003 rows=1 loops=3001)
               Index Cond: (unique1 = o1.unique1)
 Planning Time: 0.376 ms
 Execution Time: 18.241 ms
(13 rows)
```

## Hash Join(Same query different Table but different Join Algorithm )- Interesting??

```
dbmsproject=# EXPLAIN ANALYZE SELECT * FROM hundredktup h1,hundredktup h2 WHERE h1.unique1 = h2.unique1 AND h1.unique1 BETWEEN 7000 AND 10000;
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=3287.27..7580.78 rows=3005 width=422) (actual time=583.417..754.851 rows=3001 loops=1)
   Hash Cond: (h2.unique1 = h1.unique1)
   ->  Seq Scan on hundredktup h2  (cost=0.00..4031.00 rows=100000 width=211) (actual time=0.749..147.503 rows=100000 loops=1)
   ->  Hash  (cost=3249.71..3249.71 rows=3005 width=211) (actual time=582.579..582.580 rows=3001 loops=1)
         Buckets: 4096  Batches: 1  Memory Usage: 745kB
         ->  Bitmap Heap Scan on hundredktup h1  (cost=75.09..3249.71 rows=3005 width=211) (actual time=5.450..578.226 rows=3001 loops=1)
               Recheck Cond: ((unique1 >= 7000) AND (unique1 <= 10000))
               Heap Blocks: exact=1914
               ->  Bitmap Index Scan on hundredktup_unique1_key  (cost=0.00..74.34 rows=3005 width=0) (actual time=5.188..5.189 rows=3001 loops=1)
                     Index Cond: ((unique1 >= 7000) AND (unique1 <= 10000))
 Planning Time: 14.833 ms
 Execution Time: 755.256 ms
(12 rows)
```

# Results

| Exp no | Hash Join | Sort Merge Join | Nested Loop Join |
|---|---|---|---|
| 1 | 744.261 | 15.581 | 1943.514 |
| 2 | 745.843 | 17.656 | 1946.19 |
| 3 | 744.190 | 15.965 | 1946.495 |
| 4 | 739.282 | 15.581 | 1941.098 |
| 5 | 755.256 | 14.517 | 1942.297 |
| **Average** | 745.664 | 15.86 | 1943.9188 |



# Results and Lessons Learned

According to our expectations sort merge join executes the fastest and is used when we used the clustered attribute of both the tables as the join predicate

Hash Join is the next best when it comes to execution time and is used when we perform a join on two tables and the join predicate is an unclustered index

The worst in performance is the nested loop join and is used when we perform a join on two very large tables and the join predicate is a range query. Even though we have an unclustered index on the table we saw the query optimizer perform a nested loop join.

# Conclusion

- We have learned how just changing a single parameter can have an impact on performance. For example, a clustered index may take less time to fetch queries than an unclustered index but, again the reverse is possible too. Forcing the query optimizer to pick a plan than what it originally did sometimes enhanced the performance and it was interesting to analyze why the optimizer would have chosen a particular plan.

- We understood the real world impacts of a clustered index vs a non-clustered index and why a good index always speeds up your execution time.

- We understood multiple parameters that postgresql has and understood the practical benefits of tuning these parameters to get better results.

# Conclusion

- We would have loved to perform speedup experiments but we didn't have that many processors at our disposal to perform speedup experiments.

- With time what we have observed is that the performance of the single user systems has not changed drastically by enabling or disabling any parameter but even those slight differences would become big ones in a multiuser environment. And hence we understood why gaining the most optimal performance on a single user system is more beneficial as it would probably be an indicator to how the system would perform in a multi-user environment.

# Lessons Learned

- The Wisconsin benchmark and it's various criteria for measuring the performance of database management system.
- The different configuration parameters in the system
- How the performance varies with difference in selection of queries of different size, indices, etc.

# Github Link:

https://github.com/poojanalan10/dbmsImplnProject/tree/main


A BIG
Thank
You!

# REFERENCES

https://www.postgresql.org/about/

https://community.microstrategy.com/s/article/How-to-increase-the-work-mem-setting-in-your-PostgreSQL-database?language=en_US

https://medium.com/we-build-state-of-the-art-software-creating/why-should-i-use-postgresql-as-database-in-my-startup-company-96de2fd375a9#:~:text=%E2%80%9CPostgreSQL%20comes%20with%20many%20features,source%2C%20PostgreSQL%20is%20highly%20extensible.