CS 587 – Database Management System Implementation Spring 2019

Team: Divya Sravani Vukkusila, Shweta Korulkar

DB of choice: PostgreSQL

- We chose to evaluate the script to populate on our local instance of PostGres Database.
- Due to our familiarity with PostGres local installation, interactions and diagnosis, we chose this option. The few advantages of PostGres is another reason, we chose this option.
- With local Instance, we would have all the flexibility over tuning or controlling all the aspects of the DB compared to a cloud hosted or a remotely hosted PostGres.
- Postgres provides better language support for Python and flexibility for user defined data types for any complex computation to be foreseen.
- PostgreSQL's query optimizer is efficient comparative to many others.
- It supports materialized views and temporary tables.

Approach:

We ran queries on three relations:

- onemilliontup1: 1000,000 tuples
- onemilliontup2: 1000,000 tuples
- tenktup: 10,000 tuples

Version: PostgreSQL 11

Each query is executed 5 times among which we dropped the fastest and slowest times and took the average of the remaining 3. We then try to analyze the result obtained and compare with expected results. If they are different we analyzed and inferred what could be reason for this variation.

Goals:

Evaluate PostgreSQL with different configuration parameter values and optimizer options and how they influence query plans chosen by query optimizer.

- Sequential Scan VS Clustered VS Non-Clustered Index Scan with enable_seqscan, enable_indexscan
- Hashing and Grouping Aggregation with enable_hashagg
- Join Algorithms with enable_mergejoin, enable_hashjoin, enable_nestloop
- work_mem impact on Merge sort, DISTINCE, ORDER BY clauses that require sorting operation

Experiment 1 – Sequential Scan and Index Scan (Clustered)

Query 1:

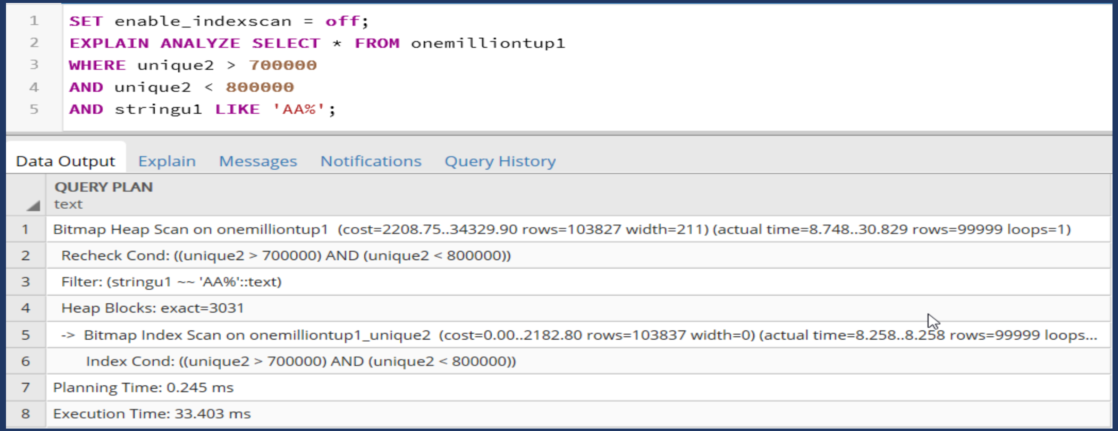
```
SET enable_indexscan = on;  // with clustered index
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique2 > 700000
AND unique2 < 800000
AND stringu1 LIKE 'AA%’;
```

- The goal is to compare performance (execution times) of Sequential full scans and index scans with nearly 10% selectivity (99999 rows out of 10,00,000 rows) with disabling and enabling enable_indexscan parameter.
- The results obtained were different from expected results.

Expected Result: When enable_indexscan = off, it is expected to execute with Sequential full scan and take more time than index scan when enable_indexscan = on.
Result Obtained: The obtained result is different as the query is executed with Bitmap Heap scan and Bitmap Index scan and it’s execution time is almost equal to index scan.
Inference: Bitmap scan occurs when select reads too less for sequential scan but too large for index scan. Since the selectivity is 10%, it could be too small for sequential scan and hence even after disabling enable_indexscan, it would have executed with bitmap scan.

Average Execution time with enable_indexscan as off = 30.13 ms
Average Execution time with enable_indexscan as on = 30.43 ms

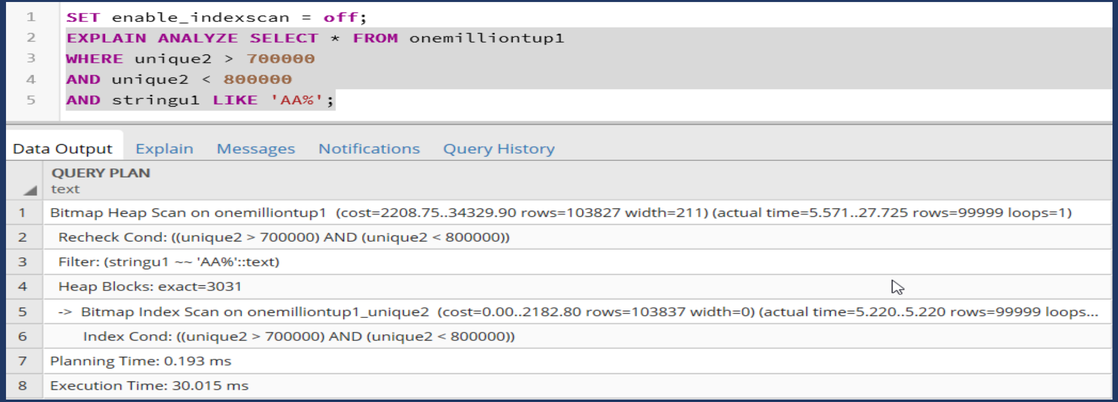
Execution times with enable_indexscan = off



```
SET enable_indexscan = off;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique2 > 700000
AND unique2 < 800000
AND stringu1 LIKE 'AA%';
```

Data Output Explain Messages Notifications Query History

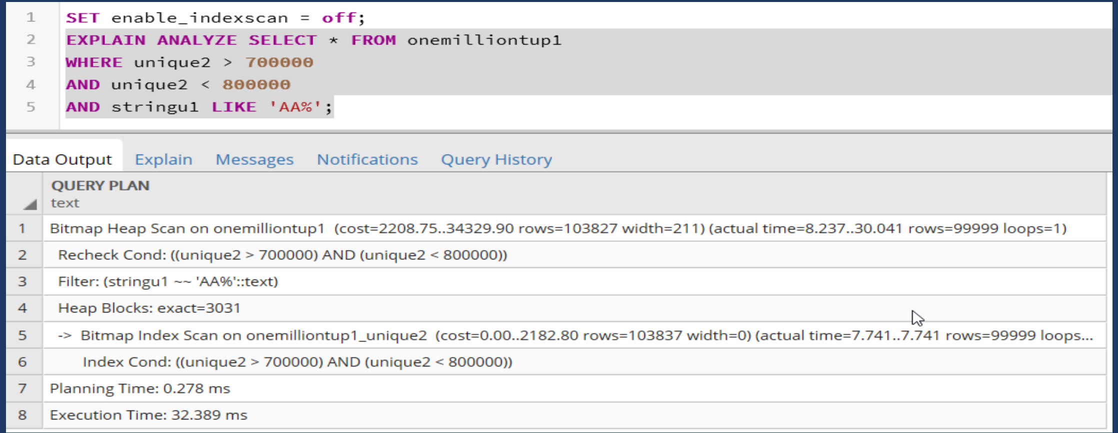
```
QUERY PLAN
text
Bitmap Heap Scan on onemilliontup1  (cost=2208.75..34329.90 rows=103827 width=211) (actual time=8.748..30.829 rows=99999 loops=1)
  Recheck Cond: ((unique2 > 700000) AND (unique2 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
  Heap Blocks: exact=3031
  -> Bitmap Index Scan on onemilliontup1_unique2  (cost=0.00..2182.80 rows=103837 width=0) (actual time=8.258..8.258 rows=99999 loops...
       Index Cond: ((unique2 > 700000) AND (unique2 < 800000))
Planning Time: 0.245 ms
Execution Time: 33.403 ms
```



```
SET enable_indexscan = off;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique2 > 700000
AND unique2 < 800000
AND stringu1 LIKE 'AA%';
```

Data Output Explain Messages Notifications Query History

```
QUERY PLAN
text
Bitmap Heap Scan on onemilliontup1  (cost=2208.75..34329.90 rows=103827 width=211) (actual time=5.571..27.725 rows=99999 loops=1)
  Recheck Cond: ((unique2 > 700000) AND (unique2 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
  Heap Blocks: exact=3031
  -> Bitmap Index Scan on onemilliontup1_unique2  (cost=0.00..2182.80 rows=103837 width=0) (actual time=5.220..5.220 rows=99999 loops...
       Index Cond: ((unique2 > 700000) AND (unique2 < 800000))
Planning Time: 0.193 ms
Execution Time: 30.015 ms
```

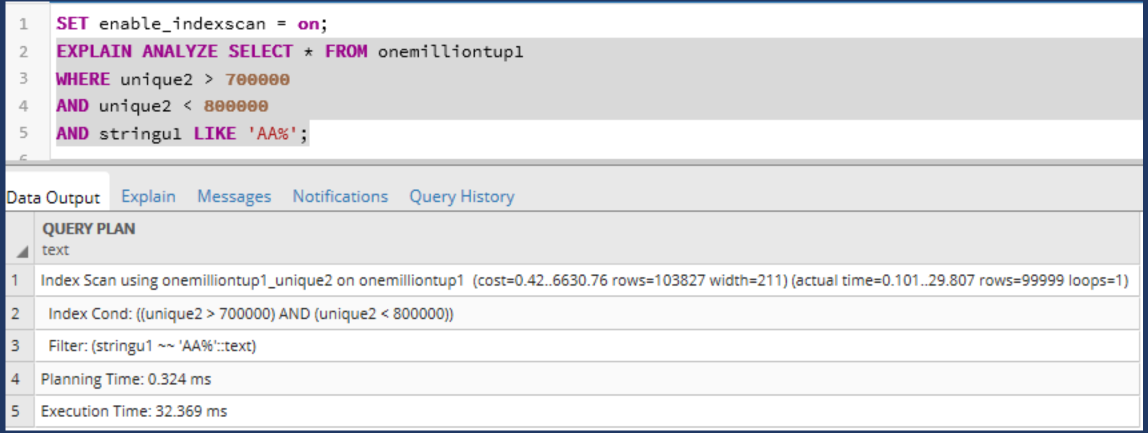


```
SET enable_indexscan = off;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique2 > 700000
AND unique2 < 800000
AND stringu1 LIKE 'AA%';
```

Data Output Explain Messages Notifications Query History

```
QUERY PLAN
text
Bitmap Heap Scan on onemilliontup1  (cost=2208.75..34329.90 rows=103827 width=211) (actual time=8.237..30.041 rows=99999 loops=1)
  Recheck Cond: ((unique2 > 700000) AND (unique2 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
  Heap Blocks: exact=3031
  -> Bitmap Index Scan on onemilliontup1_unique2  (cost=0.00..2182.80 rows=103837 width=0) (actual time=7.741..7.741 rows=99999 loops...
       Index Cond: ((unique2 > 700000) AND (unique2 < 800000))
Planning Time: 0.278 ms
Execution Time: 32.389 ms
```

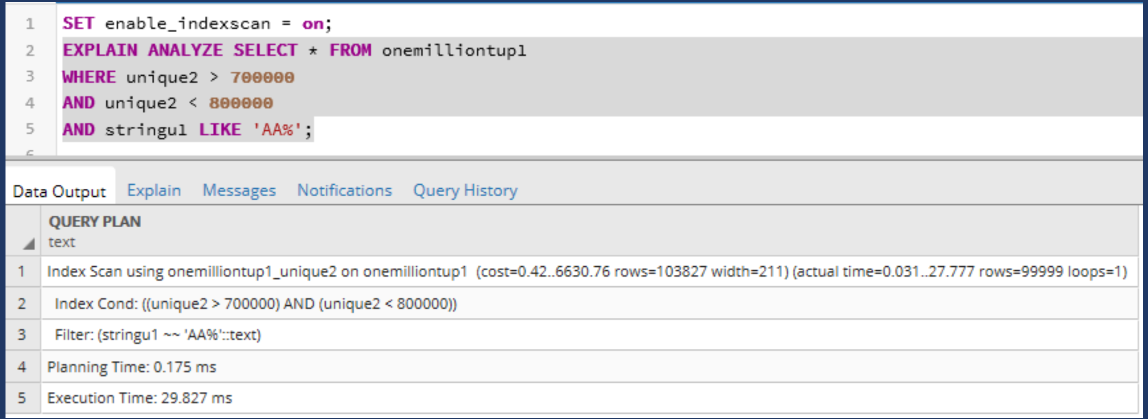
Execution times with enable_indexscan = on



```
SET enable_indexscan = on;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique2 > 700000
AND unique2 < 800000
AND stringu1 LIKE 'AA%';
```

Data Output Explain Messages Notifications Query History

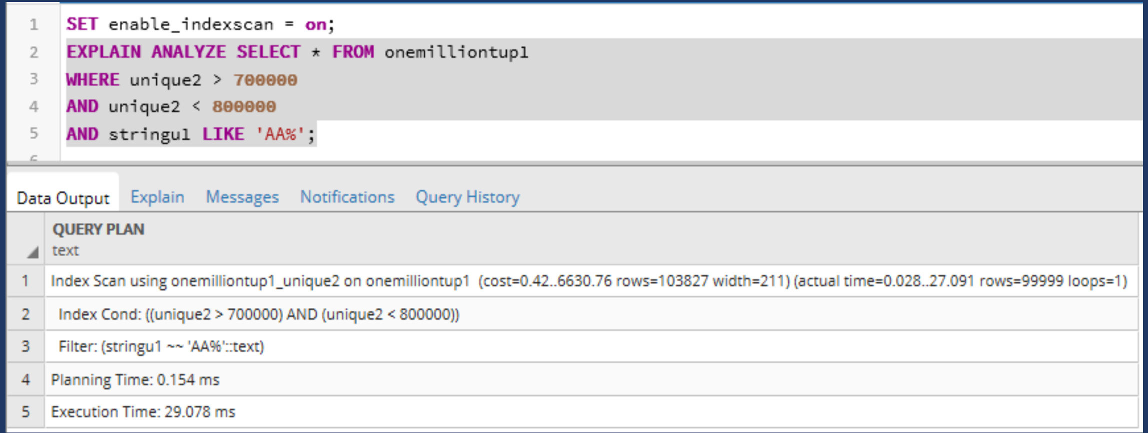
```
QUERY PLAN
text
Index Scan using onemilliontup1_unique2 on onemilliontup1  (cost=0.42..6630.76 rows=103827 width=211) (actual time=0.101..29.807 rows=99999 loops=1)
  Index Cond: ((unique2 > 700000) AND (unique2 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
Planning Time: 0.324 ms
Execution Time: 32.369 ms
```



```
SET enable_indexscan = on;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique2 > 700000
AND unique2 < 800000
AND stringu1 LIKE 'AA%';
```

Data Output Explain Messages Notifications Query History

```
QUERY PLAN
text
Index Scan using onemilliontup1_unique2 on onemilliontup1  (cost=0.42..6630.76 rows=103827 width=211) (actual time=0.031..27.777 rows=99999 loops=1)
  Index Cond: ((unique2 > 700000) AND (unique2 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
Planning Time: 0.175 ms
Execution Time: 29.827 ms
```



```
SET enable_indexscan = on;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique2 > 700000
AND unique2 < 800000
AND stringu1 LIKE 'AA%';
```

Data Output Explain Messages Notifications Query History

```
QUERY PLAN
text
Index Scan using onemilliontup1_unique2 on onemilliontup1  (cost=0.42..6630.76 rows=103827 width=211) (actual time=0.028..27.091 rows=99999 loops=1)
  Index Cond: ((unique2 > 700000) AND (unique2 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
Planning Time: 0.154 ms
Execution Time: 29.078 ms
```

Experiment 1 – Clustered and Non-Clustered Index Scan

Query 2:

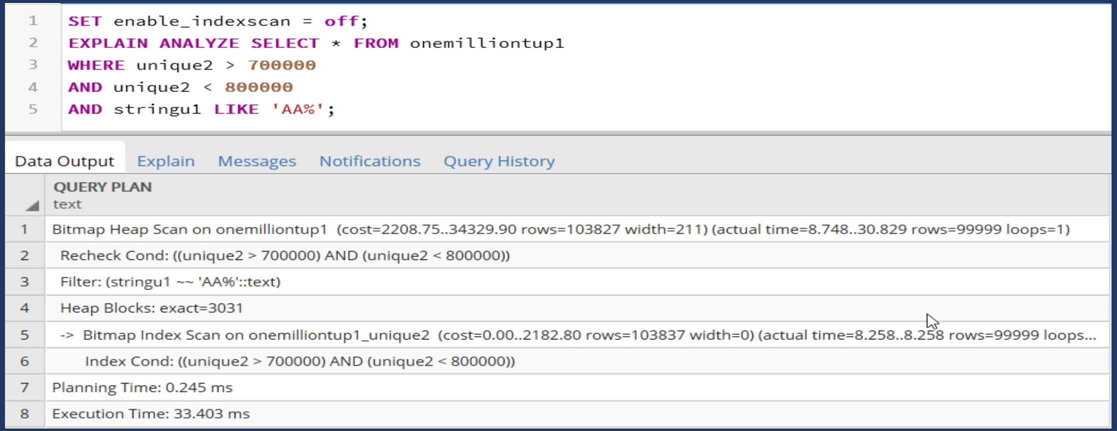
```
CREATE INDEX onemilliontup1_unique1 ON onemilliontup1(unique1);
SET enable_indexscan = on; // with non-clustered index
SET enable_seqscan = off;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique1 > 500000
AND unique1 < 600000
AND stringu1 LIKE 'AA%';
```

- The goal is to compare performance (execution times) of clustered and non-clustered index scans with nearly 10% selectivity (99999 rows out of 10,00,000 rows) with enabling enable_indexscan parameter.
- The results obtained were partially different from expected results.

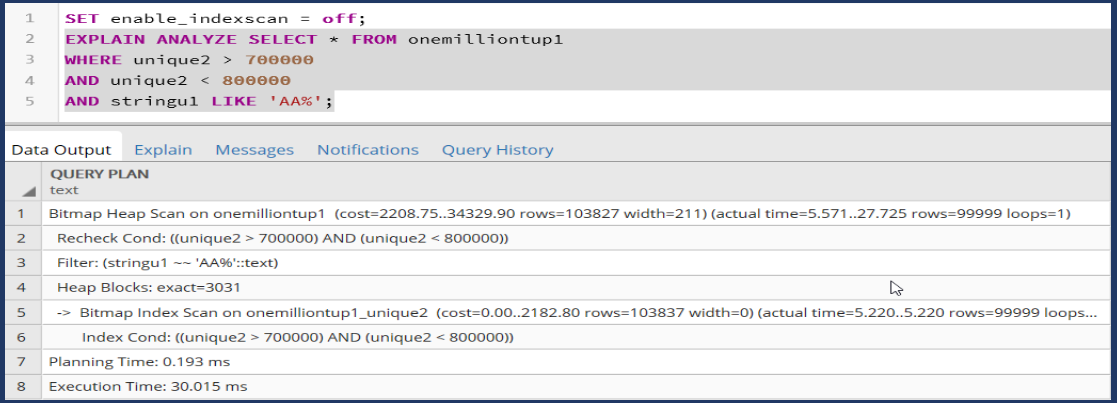
Expected Result: As there is unclustered index on unique1 column, the query is expected to execute with non-clustered index scan since selectivity is slightly lesser than 10% and with more execution time than clustered index scan.
Result Obtained: The obtained result is partially different in a way that the query is executed with Bitmap Heap scan and Bitmap Index scan instead of non-clustered index on unique1 and as expected the execution time with clustered index is far lesser than bitmap heap scan.

Average Execution time with clustered index scan = 30.43 ms
Average Execution time with non-clustered index scan = 529 ms

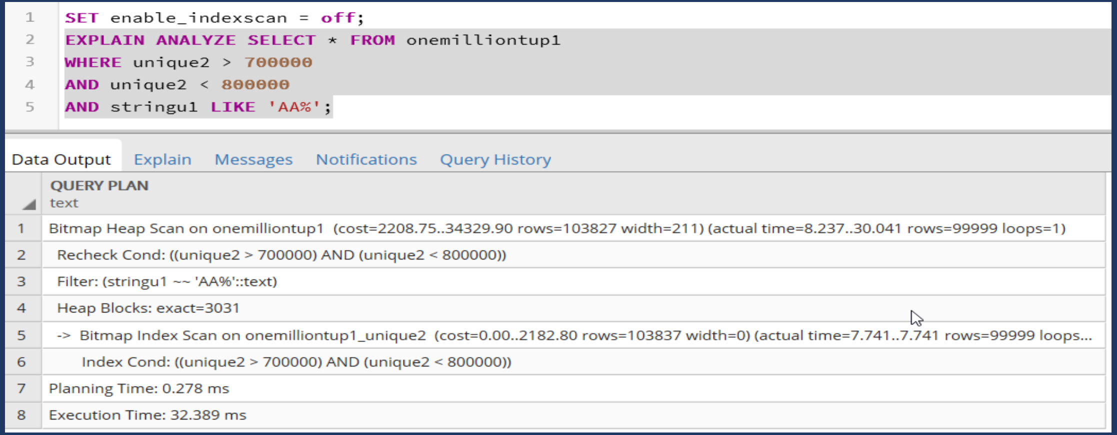
Execution times with clustered index scan


```sql
SET enable_indexscan = off;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique2 > 700000
AND unique2 < 800000
AND stringu1 LIKE 'AA%';
```

```
QUERY PLAN
text
Bitmap Heap Scan on onemilliontup1  (cost=2208.75..34329.90 rows=103827 width=211) (actual time=8.748..30.829 rows=99999 loops=1)
  Recheck Cond: ((unique2 > 700000) AND (unique2 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
  Heap Blocks: exact=3031
  ->  Bitmap Index Scan on onemilliontup1_unique2  (cost=0.00..2182.80 rows=103837 width=0) (actual time=8.258..8.258 rows=99999 loops...
        Index Cond: ((unique2 > 700000) AND (unique2 < 800000))
Planning Time: 0.245 ms
Execution Time: 33.403 ms
```

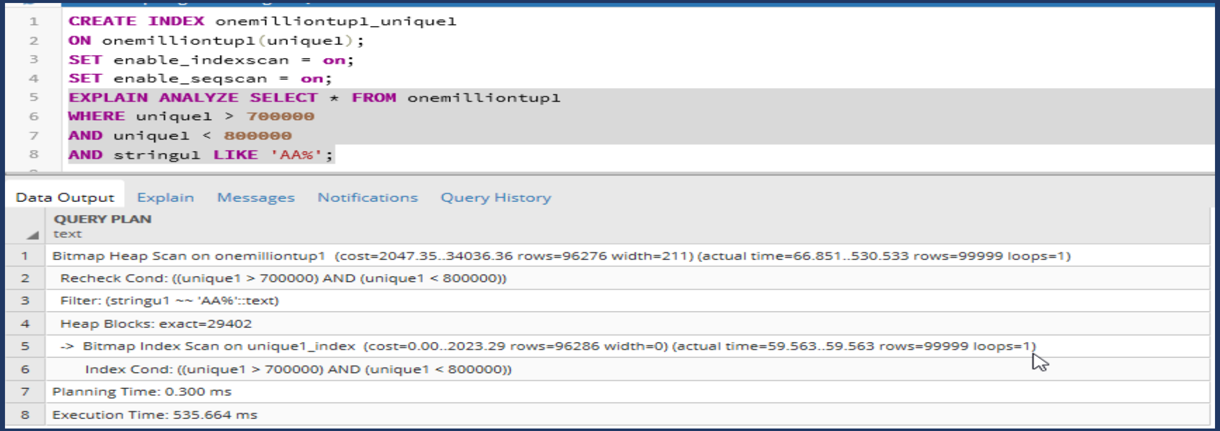

```sql
SET enable_indexscan = off;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique2 > 700000
AND unique2 < 800000
AND stringu1 LIKE 'AA%';
```

```
QUERY PLAN
text
Bitmap Heap Scan on onemilliontup1  (cost=2208.75..34329.90 rows=103827 width=211) (actual time=5.571..27.725 rows=99999 loops=1)
  Recheck Cond: ((unique2 > 700000) AND (unique2 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
  Heap Blocks: exact=3031
  ->  Bitmap Index Scan on onemilliontup1_unique2  (cost=0.00..2182.80 rows=103837 width=0) (actual time=5.220..5.220 rows=99999 loops...
        Index Cond: ((unique2 > 700000) AND (unique2 < 800000))
Planning Time: 0.193 ms
Execution Time: 30.015 ms
```

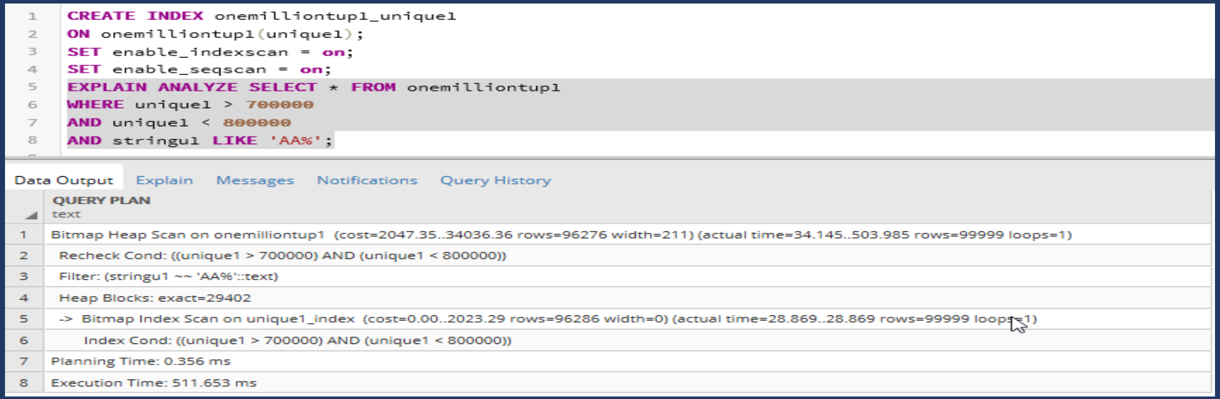

```sql
SET enable_indexscan = off;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique2 > 700000
AND unique2 < 800000
AND stringu1 LIKE 'AA%';
```

```
QUERY PLAN
text
Bitmap Heap Scan on onemilliontup1  (cost=2208.75..34329.90 rows=103827 width=211) (actual time=8.237..30.041 rows=99999 loops=1)
  Recheck Cond: ((unique2 > 700000) AND (unique2 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
  Heap Blocks: exact=3031
  ->  Bitmap Index Scan on onemilliontup1_unique2  (cost=0.00..2182.80 rows=103837 width=0) (actual time=7.741..7.741 rows=99999 loops...
        Index Cond: ((unique2 > 700000) AND (unique2 < 800000))
Planning Time: 0.278 ms
Execution Time: 32.389 ms
```

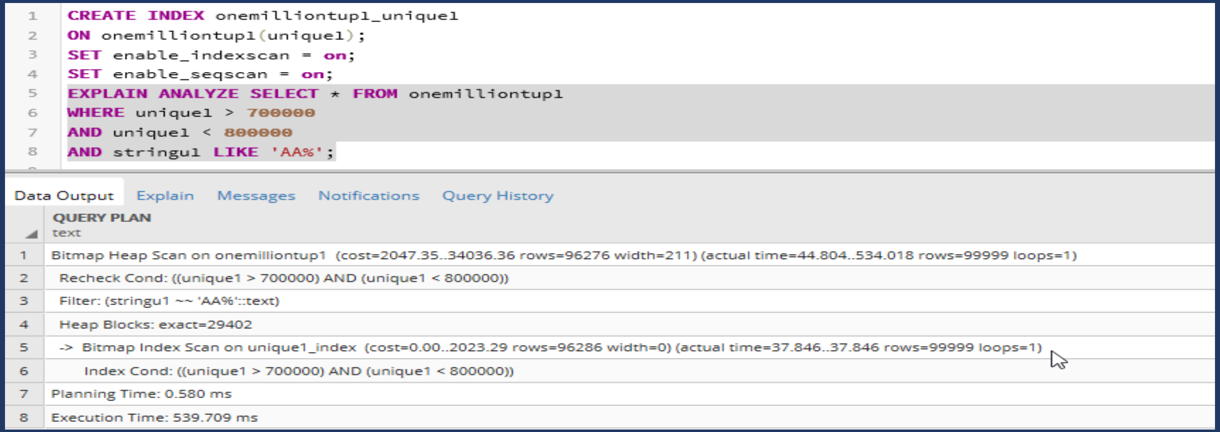
Execution times with non-clustered index scan


```sql
CREATE INDEX onemilliontup1_unique1
ON onemilliontup1(unique1);
SET enable_indexscan = on;
SET enable_seqscan = on;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique1 > 700000
AND unique1 < 800000
AND stringu1 LIKE 'AA%';
```

```
QUERY PLAN
text
Bitmap Heap Scan on onemilliontup1  (cost=2047.35..34036.36 rows=96276 width=211) (actual time=66.851..530.533 rows=99999 loops=1)
  Recheck Cond: ((unique1 > 700000) AND (unique1 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
  Heap Blocks: exact=29402
  ->  Bitmap Index Scan on unique1_index  (cost=0.00..2023.29 rows=96286 width=0) (actual time=59.563..59.563 rows=99999 loops=1)
        Index Cond: ((unique1 > 700000) AND (unique1 < 800000))
Planning Time: 0.300 ms
Execution Time: 535.664 ms
```


```sql
CREATE INDEX onemilliontup1_unique1
ON onemilliontup1(unique1);
SET enable_indexscan = on;
SET enable_seqscan = on;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique1 > 700000
AND unique1 < 800000
AND stringu1 LIKE 'AA%';
```

```
QUERY PLAN
text
Bitmap Heap Scan on onemilliontup1  (cost=2047.35..34036.36 rows=96276 width=211) (actual time=34.145..503.985 rows=99999 loops=1)
  Recheck Cond: ((unique1 > 700000) AND (unique1 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
  Heap Blocks: exact=29402
  ->  Bitmap Index Scan on unique1_index  (cost=0.00..2023.29 rows=96286 width=0) (actual time=28.869..28.869 rows=99999 loops=1)
        Index Cond: ((unique1 > 700000) AND (unique1 < 800000))
Planning Time: 0.356 ms
Execution Time: 511.653 ms
```


```sql
CREATE INDEX onemilliontup1_unique1
ON onemilliontup1(unique1);
SET enable_indexscan = on;
SET enable_seqscan = on;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1
WHERE unique1 > 700000
AND unique1 < 800000
AND stringu1 LIKE 'AA%';
```

```
QUERY PLAN
text
Bitmap Heap Scan on onemilliontup1  (cost=2047.35..34036.36 rows=96276 width=211) (actual time=44.804..534.018 rows=99999 loops=1)
  Recheck Cond: ((unique1 > 700000) AND (unique1 < 800000))
  Filter: (stringu1 ~~ 'AA%'::text)
  Heap Blocks: exact=29402
  ->  Bitmap Index Scan on unique1_index  (cost=0.00..2023.29 rows=96286 width=0) (actual time=37.846..37.846 rows=99999 loops=1)
        Index Cond: ((unique1 > 700000) AND (unique1 < 800000))
Planning Time: 0.580 ms
Execution Time: 539.709 ms
```

Experiment 2 – Hashing and Grouping Aggregation

Query 1:

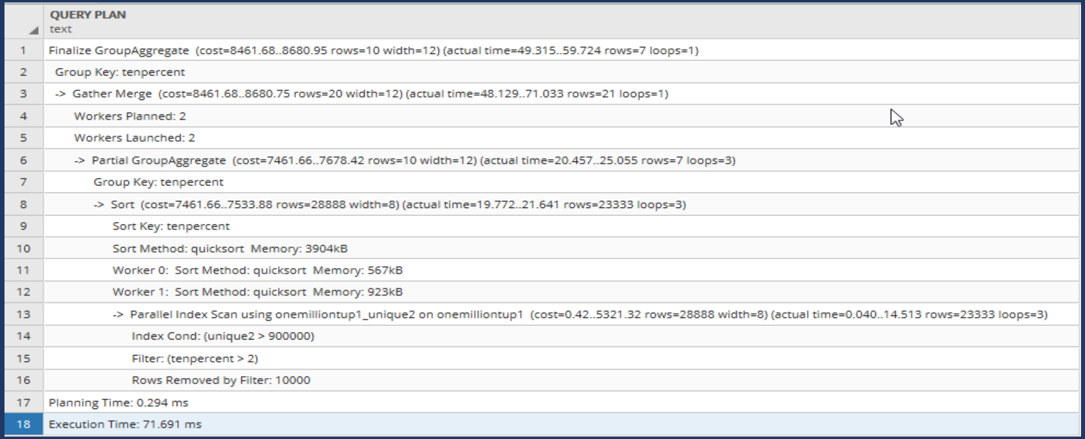
```
SET work_mem = '20MB';  // with clustered index and explicit work_mem setting
SET enable_hashagg = on;  // enable hashing aggregation
SET enable_indexscan = on;
EXPLAIN ANALYZE SELECT COUNT(unique2) FROM onemilliontup1
WHERE unique2 > 900000
GROUP BY tenPercent
HAVING tenPercent  >  2;
```

- The goal is to compare performance (execution times) of aggregation query with enabling and disabling enable_hashagg parameter.
- The results obtained were same as the expected results.

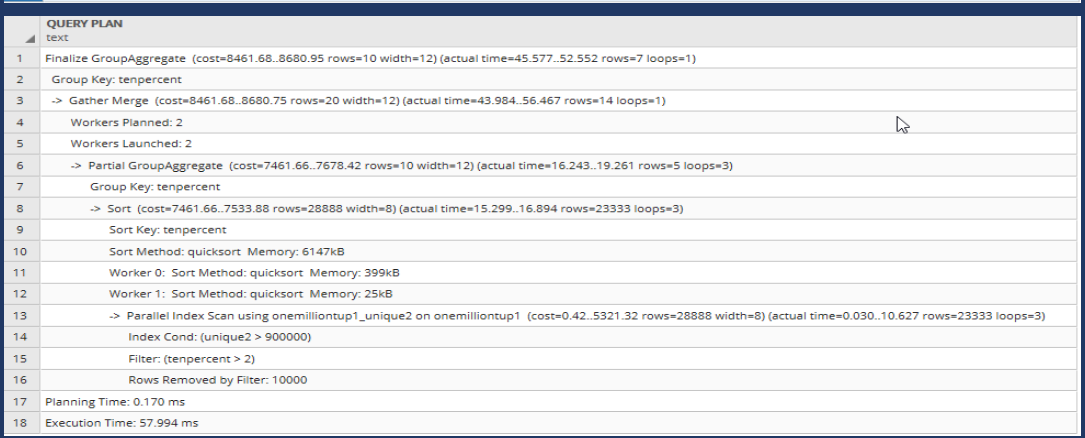
<u>Expected Result & Result Obtained:</u> The execution time of aggregation query with hashing aggregation is less than the execution time with grouping aggregation.
<u>Inference:</u> Since selectivity is slightly less than 10% and also work_mem is assigned with more value (20MB) than default value (4MB), hence it has enough memory to fit the hash table in the memory and hence hashing aggregation will have less execution time than grouping aggregation. This is because grouping aggregation will sort all the rows and then groups them accordingly.

Average Execution time with enable_hashagg disabled = 64.58 ms
Average Execution time with enable_hashagg enabled = 34.83 ms

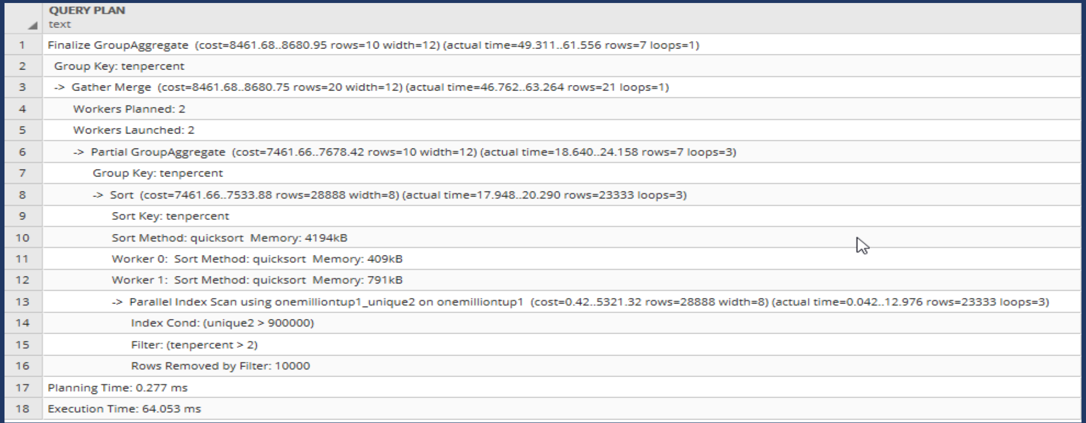
With enable_hashagg = off



```
QUERY PLAN
text
Finalize GroupAggregate  (cost=8461.68..8680.95 rows=10 width=12) (actual time=49.315..59.724 rows=7 loops=1)
  Group Key: tenpercent
  ->  Gather Merge  (cost=8461.68..8680.75 rows=20 width=12) (actual time=48.129..71.033 rows=21 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial GroupAggregate  (cost=7461.66..7678.42 rows=10 width=12) (actual time=20.457..25.055 rows=7 loops=3)
              Group Key: tenpercent
              ->  Sort  (cost=7461.66..7533.88 rows=28888 width=8) (actual time=19.772..21.641 rows=23333 loops=3)
                    Sort Key: tenpercent
                    Sort Method: quicksort  Memory: 3904kB
                    Worker 0:  Sort Method: quicksort  Memory: 567kB
                    Worker 1:  Sort Method: quicksort  Memory: 923kB
                    ->  Parallel Index Scan using onemilliontup1_unique2 on onemilliontup1  (cost=0.42..5321.32 rows=28888 width=8) (actual time=0.040..14.513 rows=23333 loops=3)
                          Index Cond: (unique2 > 900000)
                          Filter: (tenpercent > 2)
                          Rows Removed by Filter: 10000
Planning Time: 0.294 ms
Execution Time: 71.691 ms
```

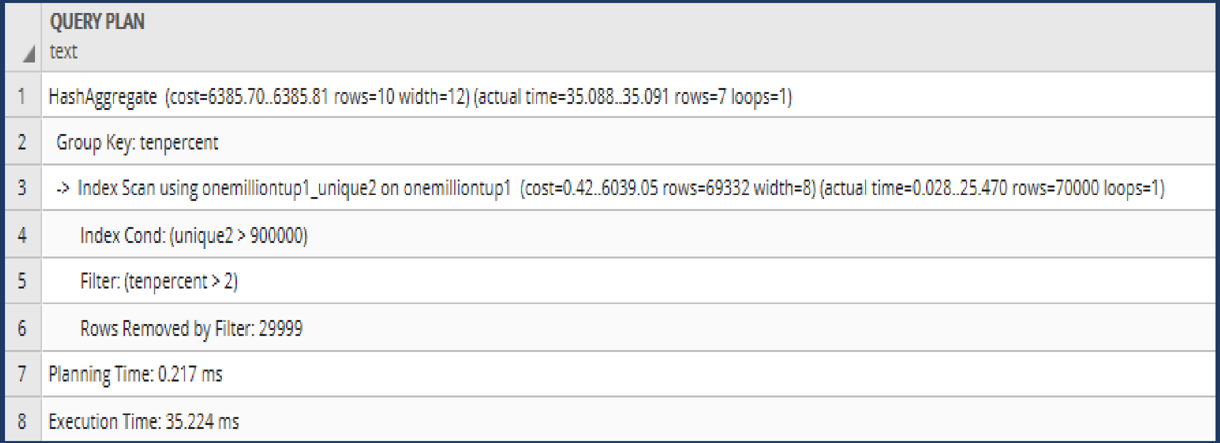


```
QUERY PLAN
text
Finalize GroupAggregate  (cost=8461.68..8680.95 rows=10 width=12) (actual time=45.577..52.552 rows=7 loops=1)
  Group Key: tenpercent
  ->  Gather Merge  (cost=8461.68..8680.75 rows=20 width=12) (actual time=43.984..56.467 rows=14 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial GroupAggregate  (cost=7461.66..7678.42 rows=10 width=12) (actual time=16.243..19.261 rows=5 loops=3)
              Group Key: tenpercent
              ->  Sort  (cost=7461.66..7533.88 rows=28888 width=8) (actual time=15.299..16.894 rows=23333 loops=3)
                    Sort Key: tenpercent
                    Sort Method: quicksort  Memory: 6147kB
                    Worker 0:  Sort Method: quicksort  Memory: 399kB
                    Worker 1:  Sort Method: quicksort  Memory: 25kB
                    ->  Parallel Index Scan using onemilliontup1_unique2 on onemilliontup1  (cost=0.42..5321.32 rows=28888 width=8) (actual time=0.030..10.627 rows=23333 loops=3)
                          Index Cond: (unique2 > 900000)
                          Filter: (tenpercent > 2)
                          Rows Removed by Filter: 10000
Planning Time: 0.170 ms
Execution Time: 57.994 ms
```

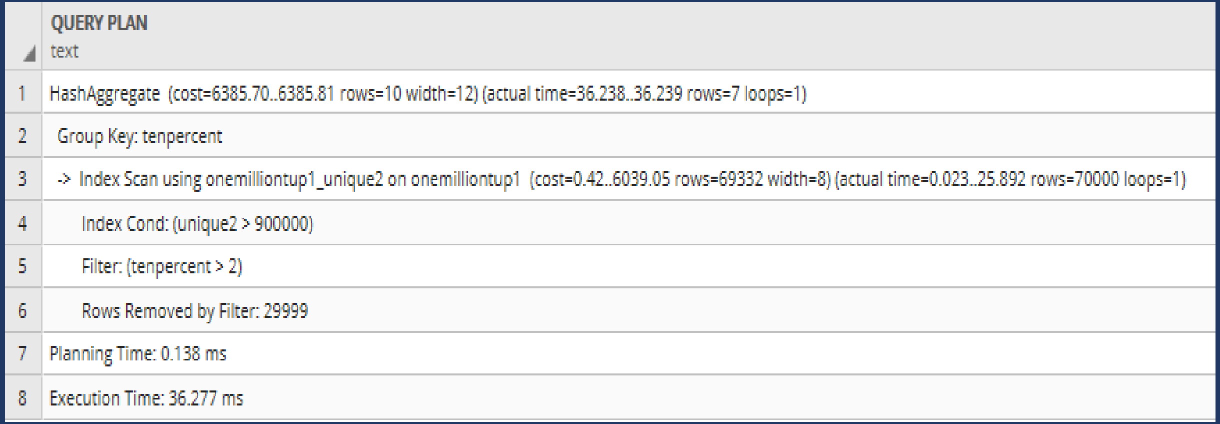


```
QUERY PLAN
text
Finalize GroupAggregate  (cost=8461.68..8680.95 rows=10 width=12) (actual time=49.311..61.556 rows=7 loops=1)
  Group Key: tenpercent
  ->  Gather Merge  (cost=8461.68..8680.75 rows=20 width=12) (actual time=46.762..63.264 rows=21 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial GroupAggregate  (cost=7461.66..7678.42 rows=10 width=12) (actual time=18.640..24.158 rows=7 loops=3)
              Group Key: tenpercent
              ->  Sort  (cost=7461.66..7533.88 rows=28888 width=8) (actual time=17.948..20.290 rows=23333 loops=3)
                    Sort Key: tenpercent
                    Sort Method: quicksort  Memory: 4194kB
                    Worker 0:  Sort Method: quicksort  Memory: 409kB
                    Worker 1:  Sort Method: quicksort  Memory: 791kB
                    ->  Parallel Index Scan using onemilliontup1_unique2 on onemilliontup1  (cost=0.42..5321.32 rows=28888 width=8) (actual time=0.042..12.976 rows=23333 loops=3)
                          Index Cond: (unique2 > 900000)
                          Filter: (tenpercent > 2)
                          Rows Removed by Filter: 10000
Planning Time: 0.277 ms
Execution Time: 64.053 ms
```

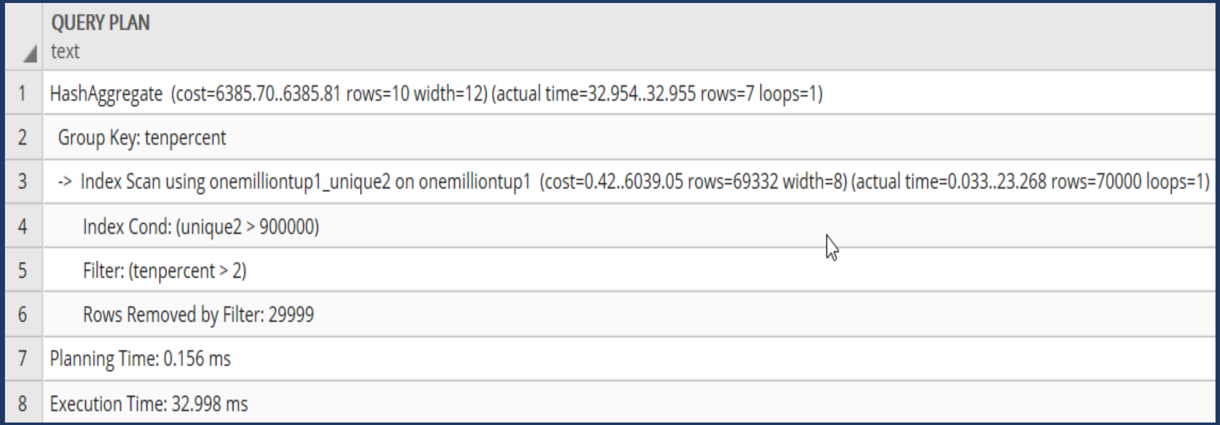
With enable_hashagg = on



```
QUERY PLAN
text
HashAggregate  (cost=6385.70..6385.81 rows=10 width=12) (actual time=35.088..35.091 rows=7 loops=1)
  Group Key: tenpercent
  ->  Index Scan using onemilliontup1_unique2 on onemilliontup1  (cost=0.42..6039.05 rows=69332 width=8) (actual time=0.028..25.470 rows=70000 loops=1)
        Index Cond: (unique2 > 900000)
        Filter: (tenpercent > 2)
        Rows Removed by Filter: 29999
Planning Time: 0.217 ms
Execution Time: 35.224 ms
```



```
QUERY PLAN
text
HashAggregate  (cost=6385.70..6385.81 rows=10 width=12) (actual time=36.238..36.239 rows=7 loops=1)
  Group Key: tenpercent
  ->  Index Scan using onemilliontup1_unique2 on onemilliontup1  (cost=0.42..6039.05 rows=69332 width=8) (actual time=0.023..25.892 rows=70000 loops=1)
        Index Cond: (unique2 > 900000)
        Filter: (tenpercent > 2)
        Rows Removed by Filter: 29999
Planning Time: 0.138 ms
Execution Time: 36.277 ms
```



```
QUERY PLAN
text
HashAggregate  (cost=6385.70..6385.81 rows=10 width=12) (actual time=32.954..32.955 rows=7 loops=1)
  Group Key: tenpercent
  ->  Index Scan using onemilliontup1_unique2 on onemilliontup1  (cost=0.42..6039.05 rows=69332 width=8) (actual time=0.033..23.268 rows=70000 loops=1)
        Index Cond: (unique2 > 900000)
        Filter: (tenpercent > 2)
        Rows Removed by Filter: 29999
Planning Time: 0.156 ms
Execution Time: 32.998 ms
```

Experiment 3 – Join Algorithms

Query 1:

```
SET enable_mergejoin = on;
SET enable_nestloop = on;
SET enable_hashjoin = on;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1 O, tenktup T
WHERE T.unique1 < 1000
AND O.unique2 = T.unique2
```

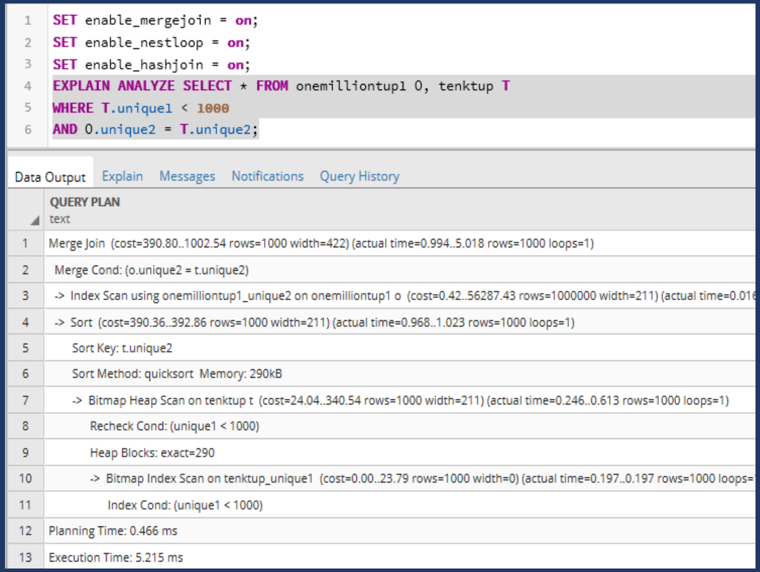
- The goal is to learn which type of join queries which joins with disabling parameters like enable_hashjoin, enable_mergejoin, enable_nestloop
- The results obtained were different from the expected results.

Expected Result: Intially all enable_hashjoin, enable_mergejoin, enable_nestloop are enabled by default. Since one of the relation is smaller (10,000 rows) and other relation is larger(1000,000 rows) with almost 100x size difference, the join query above was expected to executed with hash join algorithm or either in other case it was expected to execute with index nested loop join as there is an clustered index on unique2 column and non-clustered index on unique1 column.

Results Obtained as below:

enable_mergejoin = on (default) With enable_hashjoin = on (default) enable_nestloop = on (default) Expected: Hashjoin Result obtained: Mergejoin Average Execution time: 5.06 ms	**enable_mergejoin = off enable_hashjoin = on enable_nestloop = on Expected: Hash join Result obtained: Index nested loop Average Execution time: 2.52 ms**	**enable_mergejoin = off enable_hashjoin = off enable_nestloop = on Expected: Index nested loop Result obtained: Index nested loop Average Execution time: 240.28 ms**

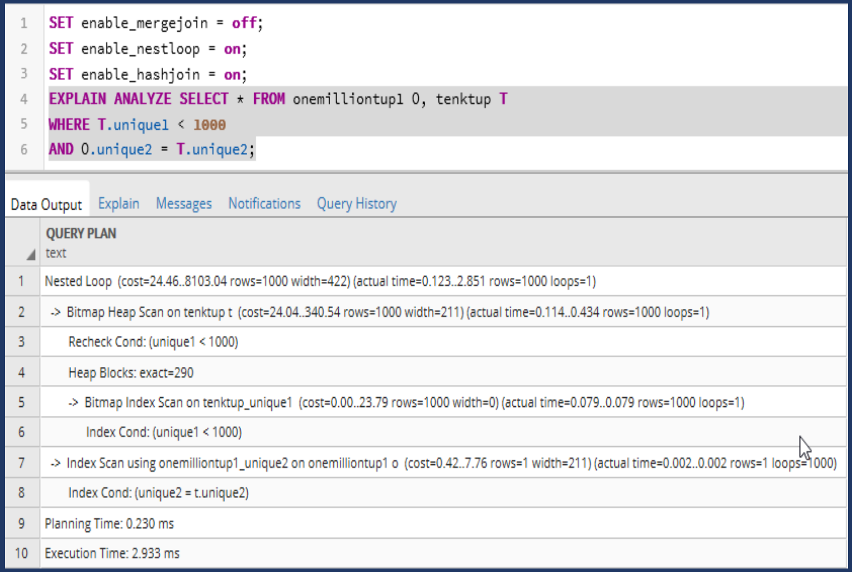
Inference: Even though one of the relation is much smaller than the other relation, the query planner has chosen merge join over hash join. This is because the execution time with merge join is far less than with hash join. This behavior could have happened because the unique2 row is primary key and hence the data is already sorted in which merge join performs more efficiently irrespective of the relations sizes. Also the execution times with merge join and index nested loop join are nearly same and since data is already on unique2, the query planner would have selected merge join over index nested loop join.



```sql
SET enable_mergejoin = on;
SET enable_nestloop = on;
SET enable_hashjoin = on;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1 O, tenktup T
WHERE T.unique1 < 1000
AND O.unique2 = T.unique2;
```

Data Output | Explain | Messages | Notifications | Query History

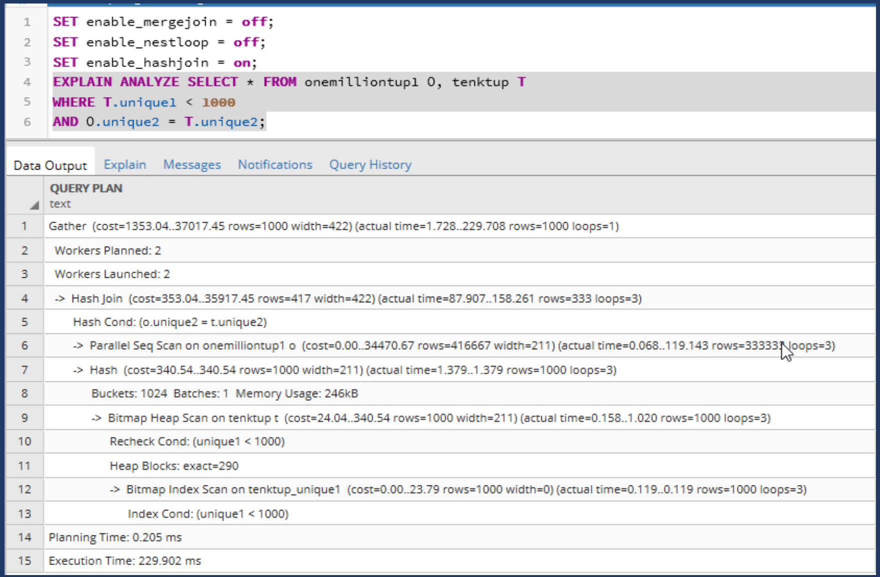
```
QUERY PLAN
text
Merge Join  (cost=390.80..1002.54 rows=1000 width=422) (actual time=0.994..5.018 rows=1000 loops=1)
  Merge Cond: (o.unique2 = t.unique2)
  ->  Index Scan using onemilliontup1_unique2 on onemilliontup1 o  (cost=0.42..56287.43 rows=1000000 width=211) (actual time=0.016
  ->  Sort  (cost=390.36..392.86 rows=1000 width=211) (actual time=0.968..1.023 rows=1000 loops=1)
        Sort Key: t.unique2
        Sort Method: quicksort  Memory: 290kB
        ->  Bitmap Heap Scan on tenktup t  (cost=24.04..340.54 rows=1000 width=211) (actual time=0.246..0.613 rows=1000 loops=1)
              Recheck Cond: (unique1 < 1000)
              Heap Blocks: exact=290
              ->  Bitmap Index Scan on tenktup_unique1  (cost=0.00..23.79 rows=1000 width=0) (actual time=0.197..0.197 rows=1000 loops=
                    Index Cond: (unique1 < 1000)
Planning Time: 0.466 ms
Execution Time: 5.215 ms
```



```sql
SET enable_mergejoin = off;
SET enable_nestloop = on;
SET enable_hashjoin = on;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1 O, tenktup T
WHERE T.unique1 < 1000
AND O.unique2 = T.unique2;
```

Data Output | Explain | Messages | Notifications | Query History

```
QUERY PLAN
text
Nested Loop  (cost=24.46..8103.04 rows=1000 width=422) (actual time=0.123..2.851 rows=1000 loops=1)
  ->  Bitmap Heap Scan on tenktup t  (cost=24.04..340.54 rows=1000 width=211) (actual time=0.114..0.434 rows=1000 loops=1)
        Recheck Cond: (unique1 < 1000)
        Heap Blocks: exact=290
        ->  Bitmap Index Scan on tenktup_unique1  (cost=0.00..23.79 rows=1000 width=0) (actual time=0.079..0.079 rows=1000 loops=1)
              Index Cond: (unique1 < 1000)
  ->  Index Scan using onemilliontup1_unique2 on onemilliontup1 o  (cost=0.42..7.76 rows=1 width=211) (actual time=0.002..0.002 rows=1 loops=1000)
        Index Cond: (unique2 = t.unique2)
Planning Time: 0.230 ms
Execution Time: 2.933 ms
```



```sql
SET enable_mergejoin = off;
SET enable_nestloop = off;
SET enable_hashjoin = on;
EXPLAIN ANALYZE SELECT * FROM onemilliontup1 O, tenktup T
WHERE T.unique1 < 1000
AND O.unique2 = T.unique2;
```

Data Output | Explain | Messages | Notifications | Query History

```
QUERY PLAN
text
Gather  (cost=1353.04..37017.45 rows=1000 width=422) (actual time=1.728..229.708 rows=1000 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Hash Join  (cost=353.04..35917.45 rows=417 width=422) (actual time=87.907..158.261 rows=333 loops=3)
        Hash Cond: (o.unique2 = t.unique2)
        ->  Parallel Seq Scan on onemilliontup1 o  (cost=0.00..34470.67 rows=416667 width=211) (actual time=0.068..119.143 rows=33333 loops=3)
        ->  Hash  (cost=340.54..340.54 rows=1000 width=211) (actual time=1.379..1.379 rows=1000 loops=3)
              Buckets: 1024  Batches: 1  Memory Usage: 246kB
              ->  Bitmap Heap Scan on tenktup t  (cost=24.04..340.54 rows=1000 width=211) (actual time=0.158..1.020 rows=1000 loops=3)
                    Recheck Cond: (unique1 < 1000)
                    Heap Blocks: exact=290
                    ->  Bitmap Index Scan on tenktup_unique1  (cost=0.00..23.79 rows=1000 width=0) (actual time=0.119..0.119 rows=1000 loops=3)
                          Index Cond: (unique1 < 1000)
Planning Time: 0.205 ms
Execution Time: 229.902 ms
```

Experiment 4 – **work_mem impact on DISTINCT & ORDER BY**

Query 1:

```
SET work_mem = '256MB'
EXPLAIN ANALYZE SELECT DISTINCT(O1.unique3)
FROM onemilliontup1 O1
ORDER BY O1.unique3
```

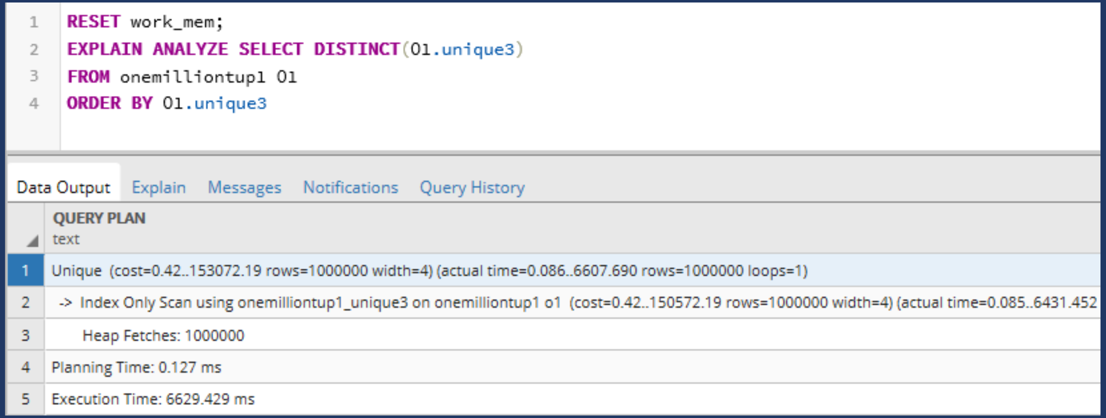
- The goal is to compare performance of query with DISTINCT AND ORDERBY clauses with varying work_mem parameter.
- The results obtained were same as the expected results.

<u>Expected Result & Result Obtained:</u> The execution time of query with DISTINCT and ORDERBY clauses with default work_mem value of 4MB should be far more than (nearly 8x times) with work_mem = 256 MB which is set explicitly.

<u>Inference:</u> work_mem specifies the amount of memory to be used by internal sort operations. Since DISTINCT and ORDERBY performs sort operations, hence assigning more memory results in lesser execution. Hence execution time for sorting queries with more memory assigned is lesser than those sorting queries with default amount of memory (4MB) assigned.

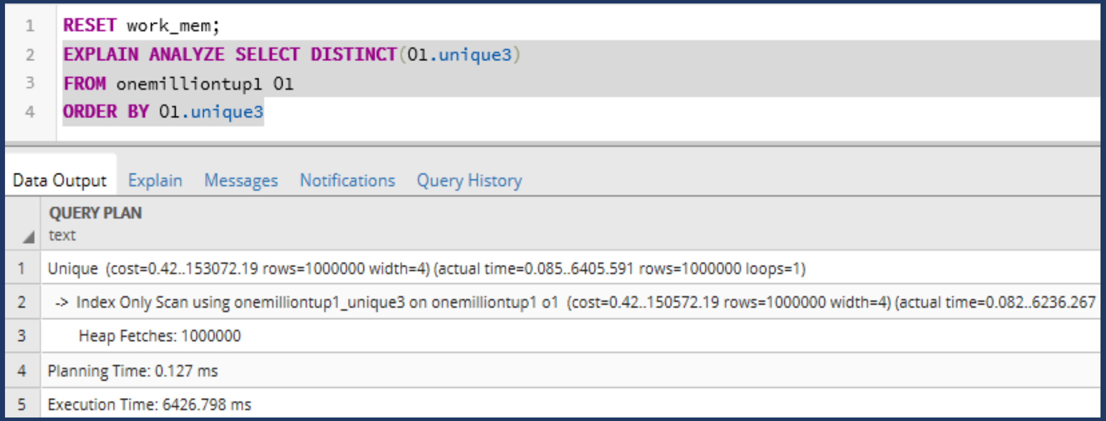
Average Execution time with default work_mem value = 6586.76 ms
Average Execution time with enable_hashagg enabled = 750.3 ms

RESET work_mem (default value = 4MB)



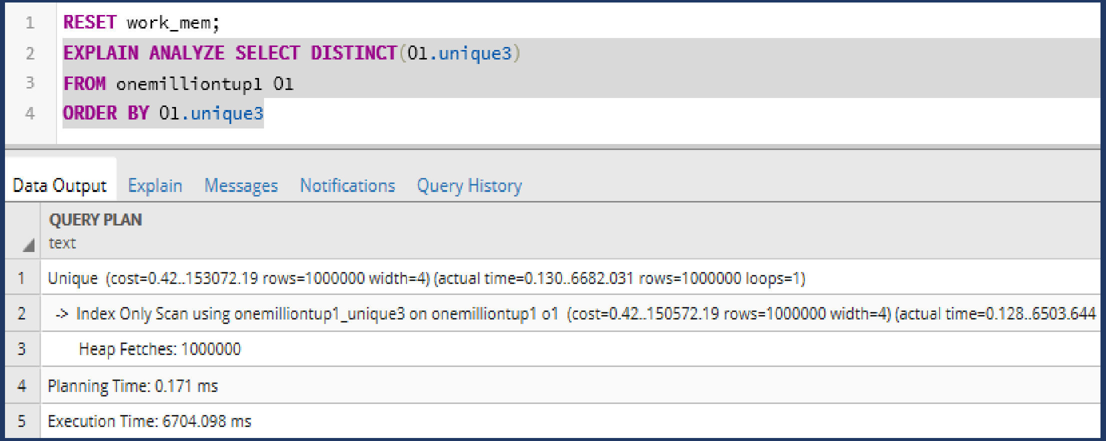
```sql
RESET work_mem;
EXPLAIN ANALYZE SELECT DISTINCT(O1.unique3)
FROM onemilliontup1 O1
ORDER BY O1.unique3
```

```
QUERY PLAN
text
Unique  (cost=0.42..153072.19 rows=1000000 width=4) (actual time=0.086..6607.690 rows=1000000 loops=1)
  -> Index Only Scan using onemilliontup1_unique3 on onemilliontup1 o1  (cost=0.42..150572.19 rows=1000000 width=4) (actual time=0.085..6431.452
       Heap Fetches: 1000000
Planning Time: 0.127 ms
Execution Time: 6629.429 ms
```



```sql
RESET work_mem;
EXPLAIN ANALYZE SELECT DISTINCT(O1.unique3)
FROM onemilliontup1 O1
ORDER BY O1.unique3
```

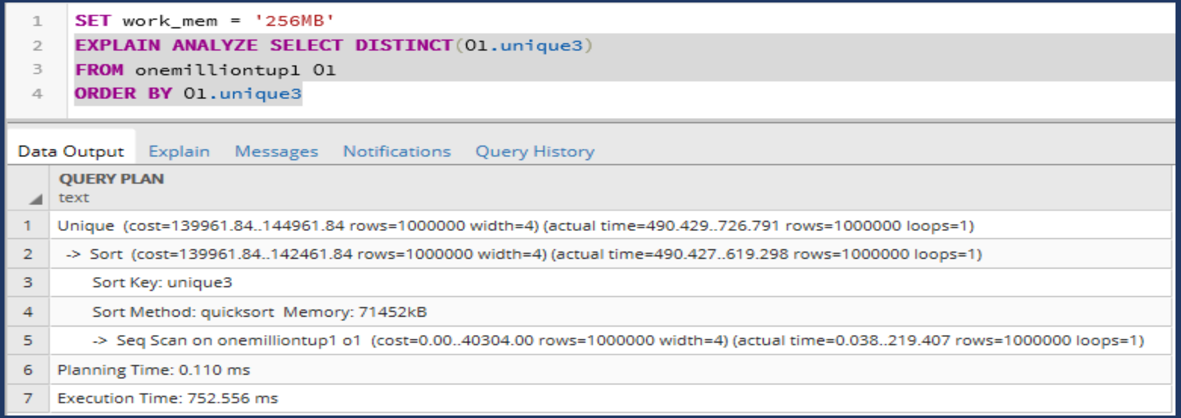
```
QUERY PLAN
text
Unique  (cost=0.42..153072.19 rows=1000000 width=4) (actual time=0.085..6405.591 rows=1000000 loops=1)
  -> Index Only Scan using onemilliontup1_unique3 on onemilliontup1 o1  (cost=0.42..150572.19 rows=1000000 width=4) (actual time=0.082..6236.267
       Heap Fetches: 1000000
Planning Time: 0.127 ms
Execution Time: 6426.798 ms
```



```sql
RESET work_mem;
EXPLAIN ANALYZE SELECT DISTINCT(O1.unique3)
FROM onemilliontup1 O1
ORDER BY O1.unique3
```

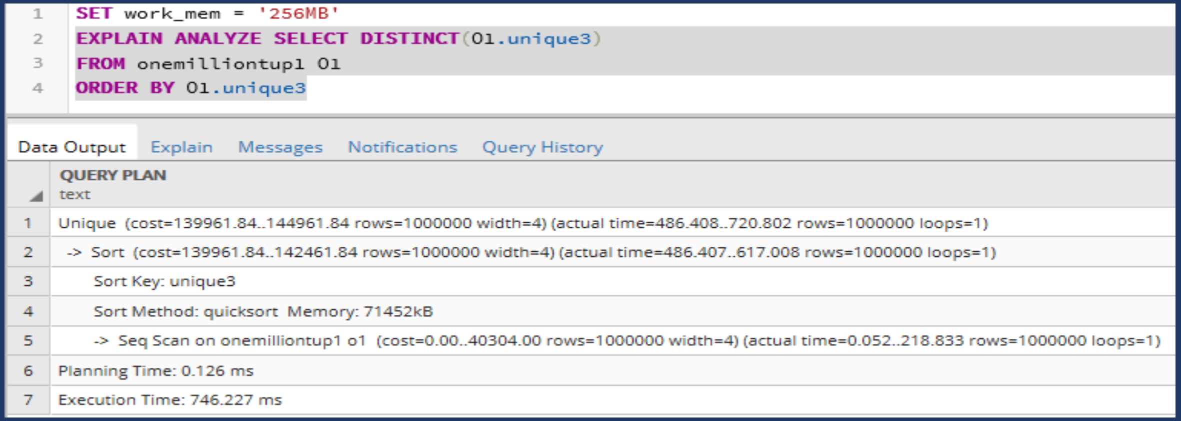
```
QUERY PLAN
text
Unique  (cost=0.42..153072.19 rows=1000000 width=4) (actual time=0.130..6682.031 rows=1000000 loops=1)
  -> Index Only Scan using onemilliontup1_unique3 on onemilliontup1 o1  (cost=0.42..150572.19 rows=1000000 width=4) (actual time=0.128..6503.644
       Heap Fetches: 1000000
Planning Time: 0.171 ms
Execution Time: 6704.098 ms
```

SET work_mem = 256MB



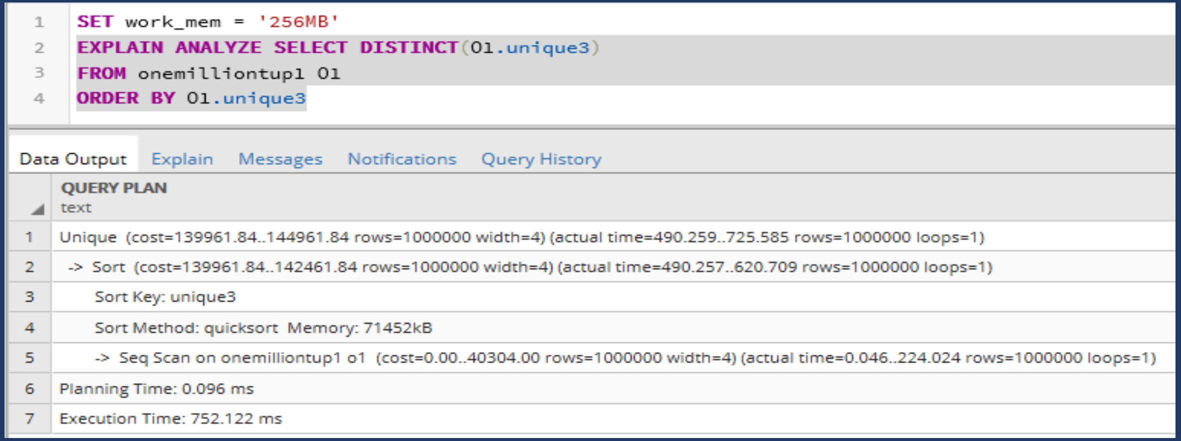
```sql
SET work_mem = '256MB'
EXPLAIN ANALYZE SELECT DISTINCT(O1.unique3)
FROM onemilliontup1 O1
ORDER BY O1.unique3
```

```
QUERY PLAN
text
Unique  (cost=139961.84..144961.84 rows=1000000 width=4) (actual time=490.429..726.791 rows=1000000 loops=1)
  -> Sort  (cost=139961.84..142461.84 rows=1000000 width=4) (actual time=490.427..619.298 rows=1000000 loops=1)
       Sort Key: unique3
       Sort Method: quicksort  Memory: 71452kB
       -> Seq Scan on onemilliontup1 o1  (cost=0.00..40304.00 rows=1000000 width=4) (actual time=0.038..219.407 rows=1000000 loops=1)
Planning Time: 0.110 ms
Execution Time: 752.556 ms
```



```sql
SET work_mem = '256MB'
EXPLAIN ANALYZE SELECT DISTINCT(O1.unique3)
FROM onemilliontup1 O1
ORDER BY O1.unique3
```

```
QUERY PLAN
text
Unique  (cost=139961.84..144961.84 rows=1000000 width=4) (actual time=486.408..720.802 rows=1000000 loops=1)
  -> Sort  (cost=139961.84..142461.84 rows=1000000 width=4) (actual time=486.407..617.008 rows=1000000 loops=1)
       Sort Key: unique3
       Sort Method: quicksort  Memory: 71452kB
       -> Seq Scan on onemilliontup1 o1  (cost=0.00..40304.00 rows=1000000 width=4) (actual time=0.052..218.833 rows=1000000 loops=1)
Planning Time: 0.126 ms
Execution Time: 746.227 ms
```



```sql
SET work_mem = '256MB'
EXPLAIN ANALYZE SELECT DISTINCT(O1.unique3)
FROM onemilliontup1 O1
ORDER BY O1.unique3
```

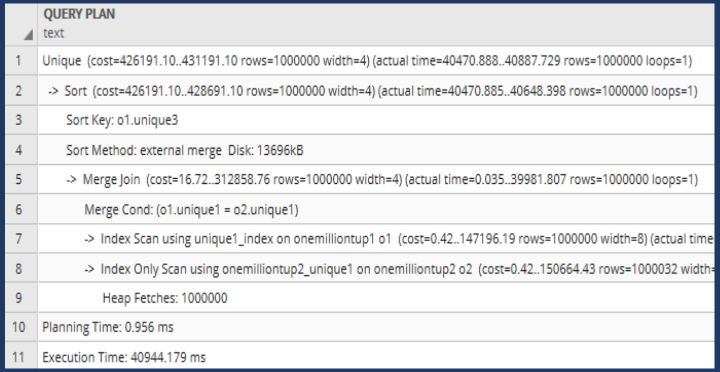
```
QUERY PLAN
text
Unique  (cost=139961.84..144961.84 rows=1000000 width=4) (actual time=490.259..725.585 rows=1000000 loops=1)
  -> Sort  (cost=139961.84..142461.84 rows=1000000 width=4) (actual time=490.257..620.709 rows=1000000 loops=1)
       Sort Key: unique3
       Sort Method: quicksort  Memory: 71452kB
       -> Seq Scan on onemilliontup1 o1  (cost=0.00..40304.00 rows=1000000 width=4) (actual time=0.046..224.024 rows=1000000 loops=1)
Planning Time: 0.096 ms
Execution Time: 752.122 ms
```

Experiment 4 – **work_mem impact on Merge sort**

Query 2:

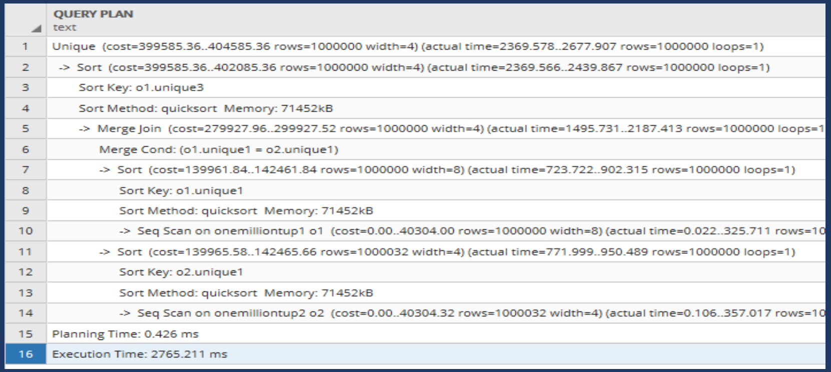
```
RESET work_mem;
SET enable_hashjoin = off;
EXPLAIN ANALYZE SELECT DISTINCT(O1.unique3)
FROM onemilliontup1 O1 INNER JOIN onemilliontup2 O2
ON O1.unique1 = O2.unique1
ORDER BY O1.unique3;
```

with work_mem = 4 MB



```
QUERY PLAN
text
Unique  (cost=426191.10..431191.10 rows=1000000 width=4) (actual time=40470.888..40887.729 rows=1000000 loops=1)
  -> Sort  (cost=426191.10..428691.10 rows=1000000 width=4) (actual time=40470.885..40648.398 rows=1000000 loops=1)
        Sort Key: o1.unique3
        Sort Method: external merge  Disk: 13696kB
        -> Merge Join  (cost=16.72..312858.76 rows=1000000 width=4) (actual time=0.035..39981.807 rows=1000000 loops=1)
              Merge Cond: (o1.unique1 = o2.unique1)
              -> Index Scan using unique1_index on onemilliontup1 o1  (cost=0.42..147196.19 rows=1000000 width=8) (actual time
              -> Index Only Scan using onemilliontup2_unique1 on onemilliontup2 o2  (cost=0.42..150664.43 rows=1000032 width=
                    Heap Fetches: 1000000
Planning Time: 0.956 ms
Execution Time: 40944.179 ms
```

with work_mem = 256 MB



```
QUERY PLAN
text
Unique  (cost=399585.36..404585.36 rows=1000000 width=4) (actual time=2369.578..2677.907 rows=1000000 loops=1)
  -> Sort  (cost=399585.36..402085.36 rows=1000000 width=4) (actual time=2369.566..2439.867 rows=1000000 loops=1)
        Sort Key: o1.unique3
        Sort Method: quicksort  Memory: 71452kB
        -> Merge Join  (cost=279927.96..299927.52 rows=1000000 width=4) (actual time=1495.731..2187.413 rows=1000000 loops=1
              Merge Cond: (o1.unique1 = o2.unique1)
              -> Sort  (cost=139961.84..142461.84 rows=1000000 width=8) (actual time=723.722..902.315 rows=1000000 loops=1)
                    Sort Key: o1.unique1
                    Sort Method: quicksort  Memory: 71452kB
                    -> Seq Scan on onemilliontup1 o1  (cost=0.00..40304.00 rows=1000000 width=8) (actual time=0.022..325.711 rows=10
              -> Sort  (cost=139965.58..142465.66 rows=1000032 width=4) (actual time=771.999..950.489 rows=1000000 loops=1)
                    Sort Key: o2.unique1
                    Sort Method: quicksort  Memory: 71452kB
                    -> Seq Scan on onemilliontup2 o2  (cost=0.00..40304.32 rows=1000032 width=4) (actual time=0.106..357.017 rows=10
Planning Time: 0.426 ms
Execution Time: 2765.211 ms
```

- The goal is to compare performance of query with Merge sort with varying work_mem parameter.
- The results obtained were same as the expected results.

Expected Result & Result Obtained: The execution time of query with merge sort with default work_mem value of 4MB should be far more than (nearly 8x times) with work_mem = 256 MB which is set explicitly.

Inference: work_mem specifies the amount of memory to be used by internal sort operations. Since merge sort performs sort operations, hence assigning more memory results in lesser execution. Hence execution time for sorting queries with more memory assigned is lesser than those sorting queries with default amount of memory (4MB) assigned.

Average Execution time with default work_mem value = ms
Average Execution time with enable_hashagg enabled = 750.3 ms

Summary:

Through four experiments after running different types of queries we have inferred few points:

- From experiment 1, we observed that Bitmap scan occurs when select reads too less for sequential scan but too large for index scan. Since the selectivity is 10%, it could be too small for sequential scan and hence even after disabling enable_indexscan, it would have executed with bitmap scan. Also in some instances the query is executed with Bitmap Heap scan and Bitmap Index scan instead of non-clustered index on unique1 even though selectivity is 10% and as expected the execution time with clustered index was far lesser than bitmap heap scan.
- From experiment2, we observed that when selectivity is slightly less than 10% and when work_mem is assigned with more value (20MB) than default value (4MB), hence it has enough memory to fit the hash table in the memory and hence hashing aggregation will have less execution time than grouping aggregation. This is because grouping aggregation will sort all the rows and then groups them accordingly.
- From experiment3, we observed that even though one of the relation is much smaller than the other relation, the query planner has chosen merge join over hash join. This is because the execution time with merge join is far less than with hash join. This behavior could have happened because the unique2 row is primary key and hence the data is already sorted in which merge join performs more efficiently irrespective of the relations sizes. Also the execution times with merge join and index nested loop join are nearly same and since data is already on unique2, the query planner would have selected merge join over index nested loop join.
- From experiment4, we observed that work_mem specifies the amount of memory to be used by internal sort operations. Since DISTINCT, ORDER BY, merge join performs sort operations, hence assigning more memory results in lesser execution. Hence execution time for sorting queries with more memory assigned is lesser than those sorting queries with default amount of memory (4MB) assigned.

Lessons learnt:

- While analyzing the different scans with configuration parameters learnt about Bitmap heap and index scan which was being used in some instances and how it is different from index scans we learnt in the class.
- Learnt about different configuration parameters Query Planning parameters and Resource Consumption parameters and how much they play key factor in influencing the query plans chosen by the query optimizer.
- Learnt that sometimes default plans chosen by the optimizer for a particular query are not optimal, and by changing the parameter values can sometimes result in tremendous improvement in performance.
- Analyzing the expected results of a query by applying the theoretical concepts learnt in the class might vary from result obtained when query is actually executed and observed that the system conditions are not ideal as assumed in theory and there are few factors like memory, size of relations, etc., that influence the query execution in real time.
- The project has given us chance to learn and understand query plans in detail especially the join algorithms query plans like batch sizes, disk spaces consumed etc.,
- It was fun trying around with different configuration parameters and experiencing the joy of faster execution of queries.

References:

1. The Wisconsin Benchmark: Past, Present, and Future - David J. DeWitt, Computer

Sciences Department, University of Wisconsin

2. Portland State University CS 587 course material

3. www.python.org/

4. www.postgresql.org/

5. www.postgresqltutorial.com/

6. www.postgresql.org/docs/9.4/runtime-config-resource.html

7. https://www.postgresql.org/docs/9.4/runtime-config-query.html

8. wiki.postgresql.org/wiki/Python/

9. www.cybertec-postgresql.com/en/postgresql-indexing-index-scan-vs-bitmap-scan-vs-sequential-scan-basics/