CS 587 Database Implementation

Spring 2019

Database Benchmarking Project - Part III

Pratik Kadam
Samarth Kedilaya

Why Postgres?

- It is the most advanced open source database management system. And also, we were already quite familiar with it as we also used it in the CS 586 class. It helps in giving us enterprise level performance and functions compared to any other open source database management systems with limitless possibilities.
- Scaling in another advantage in PostgreSQL. Since it is an open source database management system, there are a lot of active users who develop and propose various new modules to the community
- PostgreSQL has a great deal of ability. It underpins complex structures and a broadness of implicit and client characterized information types. It gives broad information limit and is trusted for its information honesty. You may not require the majority of the propelled highlights we've surveyed here for putting away information, however since information needs can develop rapidly, there is without a doubt, clear advantage to having everything readily available.
- The arrangement parameters give an unrefined strategy for impacting the query plans picked by the query optimizer. On the off chance that the default plan picked by the optimizer for a specific query isn't ideal, a transitory arrangement is to utilize one of these setup parameters to drive the optimizer to pick an alternate arrangement.

Objectives:-

- To learn how join algorithms are implemented in PostGreSQL.
- To learn how the query execution time varies when we vary the amount of memory used by internal sort operations and hash tables.
- To learn how sequential scan has an effect on the query execution runtime.
- To learn how the query gets affected if we stop using the bitmap scan in postgres.

Implementation:-

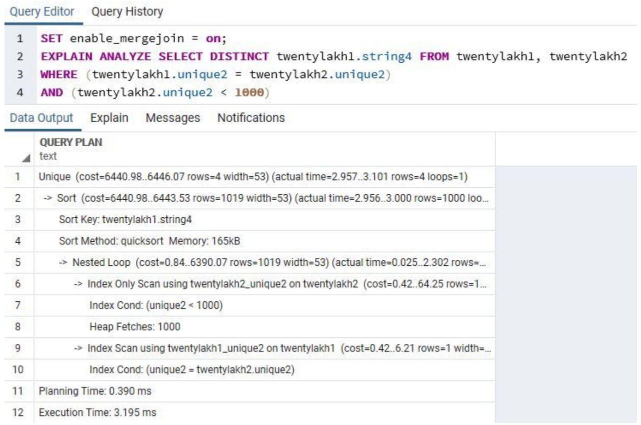
1) **System**: PostgreSQL Version 10.7
2) **Dataset**:
    a) Tenlakh1 → 1 million tuples
    b) Twentylakh1 → 2 million tuples
    c) Twentylakh2 → 2 million tuples

Experiment - 1: Join Algorithms

SQL Query: SELECT DISTINCT twentylakh1.string4 FROM twentylakh1, twentylakh2 WHERE (twentylakh1.unique2 = twentylakh2.unique2) AND (twentylakh2.unique2 < 1000)

Parameter: The parameter that we toggled here was the merge join. (SET enable_mergejoin = on). When Postgres thinks the hash table required for a hash join will surpass memory, Postgres will ordinarily utilize a merge join. A merge join ends up utilizing plate substantially more successfully than a hash join can. Like a hash join, merge join just takes a shot at joins where the join condition is an equity channel.

Expected Results & Result found: As the number of tuples in either of the tables will not fit in the memory, it will opt for merge join, which might provide the best execution time. As expected we found that the query chose to go ahead with merge join with best possible execution time of 3.195ms.



```
SET enable_mergejoin = on;
EXPLAIN ANALYZE SELECT DISTINCT twentylakh1.string4 FROM twentylakh1, twentylakh2
WHERE (twentylakh1.unique2 = twentylakh2.unique2)
AND (twentylakh2.unique2 < 1000)
```

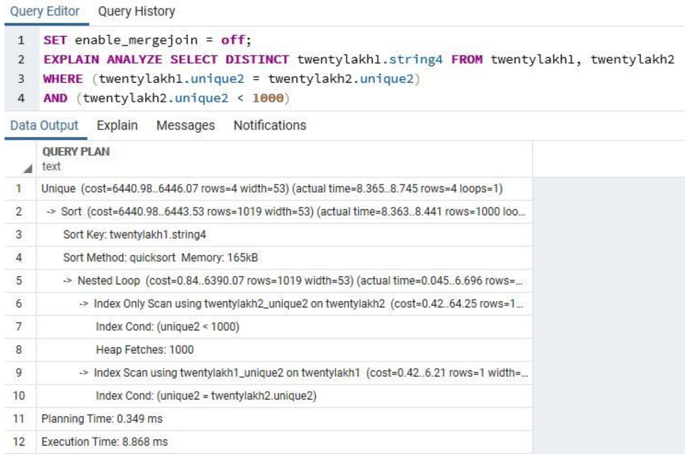
```
QUERY PLAN
text
1   Unique  (cost=6440.98..6446.07 rows=4 width=53) (actual time=2.957..3.101 rows=4 loops=1)
2     ->  Sort  (cost=6440.98..6443.53 rows=1019 width=53) (actual time=2.956..3.000 rows=1000 loo...
3           Sort Key: twentylakh1.string4
4           Sort Method: quicksort  Memory: 165kB
5           ->  Nested Loop  (cost=0.84..6390.07 rows=1019 width=53) (actual time=0.025..2.302 rows=...
6                 ->  Index Only Scan using twentylakh2_unique2 on twentylakh2  (cost=0.42..64.25 rows=1...
7                       Index Cond: (unique2 < 1000)
8                       Heap Fetches: 1000
9                 ->  Index Scan using twentylakh1_unique2 on twentylakh1  (cost=0.42..6.21 rows=1 width=...
10                      Index Cond: (unique2 = twentylakh2.unique2)
11  Planning Time: 0.390 ms
12  Execution Time: 3.195 ms
```

Experiment - 1: Join Algorithms

SQL Query: SELECT DISTINCT twentylakh1.string4 FROM twentylakh1, twentylakh2 WHERE (twentylakh1.unique2 = twentylakh2.unique2) AND (twentylakh2.unique2 < 1000)

Parameter: The parameter that we toggled here was the merge join. (SET enable_mergejoin = off). When Postgres thinks the hash table required for a hash join will surpass memory, Postgres will ordinarily utilize a merge join. A merge join ends up utilizing plate substantially more successfully than a hash join can. Like a hash join, merge join just takes a shot at joins where the join condition is an equity channel.

Expected Results & Results found: Now, if we turn off merge join, the execution time increases. It will use a nested loop.We found that the query chose nested loop increasing our query execution time to 8.868ms.


Query Editor Query History

```sql
SET enable_mergejoin = off;
EXPLAIN ANALYZE SELECT DISTINCT twentylakh1.string4 FROM twentylakh1, twentylakh2
WHERE (twentylakh1.unique2 = twentylakh2.unique2)
AND (twentylakh2.unique2 < 1000)
```

Data Output Explain Messages Notifications

```
QUERY PLAN
text
Unique  (cost=6440.98..6446.07 rows=4 width=53) (actual time=8.365..8.745 rows=4 loops=1)
  ->  Sort  (cost=6440.98..6443.53 rows=1019 width=53) (actual time=8.363..8.441 rows=1000 loo...
        Sort Key: twentylakh1.string4
        Sort Method: quicksort  Memory: 165kB
        ->  Nested Loop  (cost=0.84..6390.07 rows=1019 width=53) (actual time=0.045..6.696 rows=...
              ->  Index Only Scan using twentylakh2_unique2 on twentylakh2  (cost=0.42..64.25 rows=1...
                    Index Cond: (unique2 < 1000)
                    Heap Fetches: 1000
              ->  Index Scan using twentylakh1_unique2 on twentylakh1  (cost=0.42..6.21 rows=1 width=...
                    Index Cond: (unique2 = twentylakh2.unique2)
Planning Time: 0.349 ms
Execution Time: 8.868 ms
```

Experiment - 2: Varying work_mem

SQL Query: SELECT twentylakh1.unique1 as MINU1 FROM twentylakh1, twentylakh2 WHERE twentylakh1.unique1 = twentylakh2.unique1 AND twentylakh1.unique2 = twentylakh2.unique2 AND twentylakh1.stringu1 LIKE 'AAAB%'

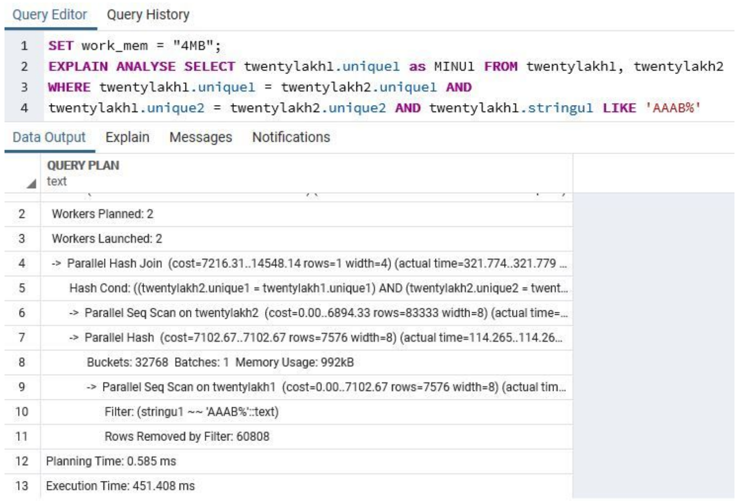
Parameter: Indicates the measure of memory to be utilized by internal sort activities and hash tables before writing to temporary disk files. Note that for a mind boggling inquiry, a few sort or hash activities may keep running in parallel; every task will be permitted to use as much memory as this esteem determines before it begins to compose information into brief records. Likewise, a few running sessions could be doing such activities simultaneously. Along these lines, the all out memory utilized could be ordinarily the estimation of work_mem; it is important to remember this reality while choosing the esteem. We varied the working memory by first setting it to 4MB and then to 128MB.

SET work_mem = “4MB”

SET work_mem = “128MB”

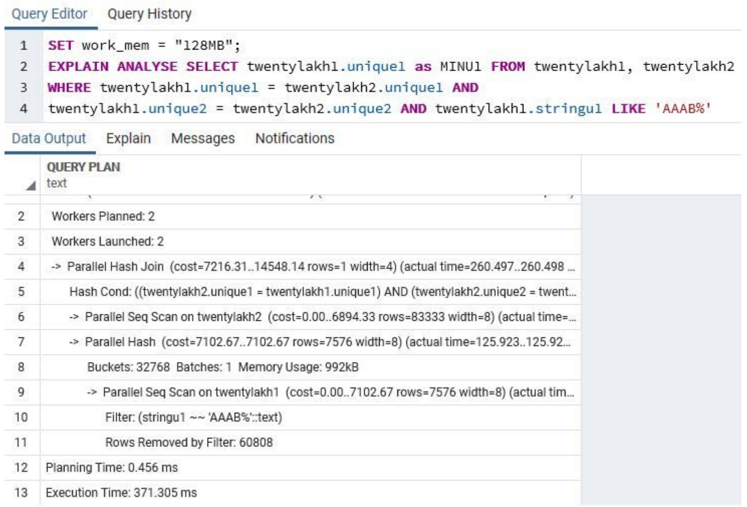
Experiment - 2: Varying work_mem

Expected Results & Results found: With the reduction of size in working memory, we found that the number of batches increased because the size of the batch is equal to the size of working memory. Therefore, when there are more number of batches, there will be more disk accesses. We found that with 4MB as our working memory, execution time took around 451.408ms whereas by increasing the working memory to 128MB, we saw a considerable reduction of our execution to 371.305ms.



```
SET work_mem = "4MB";
EXPLAIN ANALYSE SELECT twentylakh1.unique1 as MINU1 FROM twentylakh1, twentylakh2
WHERE twentylakh1.unique1 = twentylakh2.unique1 AND
twentylakh1.unique2 = twentylakh2.unique2 AND twentylakh1.stringu1 LIKE 'AAAB%'
```

```
2   Workers Planned: 2
3   Workers Launched: 2
4   -> Parallel Hash Join  (cost=7216.31..14548.14 rows=1 width=4) (actual time=321.774..321.779 ...
5        Hash Cond: ((twentylakh2.unique1 = twentylakh1.unique1) AND (twentylakh2.unique2 = twent...
6        -> Parallel Seq Scan on twentylakh2  (cost=0.00..6894.33 rows=83333 width=8) (actual time=...
7        -> Parallel Hash  (cost=7102.67..7102.67 rows=7576 width=8) (actual time=114.265..114.26...
8             Buckets: 32768  Batches: 1  Memory Usage: 992kB
9             -> Parallel Seq Scan on twentylakh1  (cost=0.00..7102.67 rows=7576 width=8) (actual tim...
10                 Filter: (stringu1 ~~ 'AAAB%'::text)
11                 Rows Removed by Filter: 60808
12  Planning Time: 0.585 ms
13  Execution Time: 451.408 ms
```



```
SET work_mem = "128MB";
EXPLAIN ANALYSE SELECT twentylakh1.unique1 as MINU1 FROM twentylakh1, twentylakh2
WHERE twentylakh1.unique1 = twentylakh2.unique1 AND
twentylakh1.unique2 = twentylakh2.unique2 AND twentylakh1.stringu1 LIKE 'AAAB%'
```

```
2   Workers Planned: 2
3   Workers Launched: 2
4   -> Parallel Hash Join  (cost=7216.31..14548.14 rows=1 width=4) (actual time=260.497..260.498 ...
5        Hash Cond: ((twentylakh2.unique1 = twentylakh1.unique1) AND (twentylakh2.unique2 = twent...
6        -> Parallel Seq Scan on twentylakh2  (cost=0.00..6894.33 rows=83333 width=8) (actual time=...
7        -> Parallel Hash  (cost=7102.67..7102.67 rows=7576 width=8) (actual time=125.923..125.92...
8             Buckets: 32768  Batches: 1  Memory Usage: 992kB
9             -> Parallel Seq Scan on twentylakh1  (cost=0.00..7102.67 rows=7576 width=8) (actual tim...
10                 Filter: (stringu1 ~~ 'AAAB%'::text)
11                 Rows Removed by Filter: 60808
12  Planning Time: 0.456 ms
13  Execution Time: 371.305 ms
```

Experiment - 3: Sequential Scan

SQL Query: SELECT tl1.twenty FROM tenlakh1 tl1 WHERE EXISTS (SELECT DISTINCT tl2.ten FROM twentylakh2 tl2 WHERE tl2.unique1 BETWEEN 250 AND 750 AND tl1.two = tl2.two)

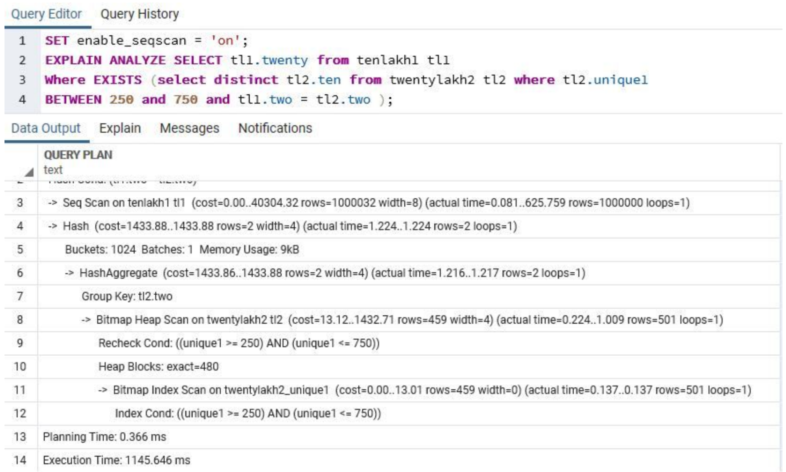
Parameter: Enables or disables the query planner's use of sequential scan plan types. It is impossible to suppress sequential scans entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is on. There are two primary reasons why Postgres will execute sequential scans:

- The first is that sequential scan is always conceivable. Regardless of what the schema of the table is, or what indices are on the table, Postgres always has the alternative of running a sequential scan.
- The other principle reason is that at times a sequential scan is in reality quicker than other existing different choices. When reading data from a disk, reading data sequentially is generally quicker than reading the data in a random order. If a large percent of the table is returned by a query, a sequential scan is usually always the better option to use because a sequential scan performs sequential I/O whereas the other alternatives available most probably will perform random I/O.

SET enable_seqscan = 'on' | SET enable_seqscan = 'off'

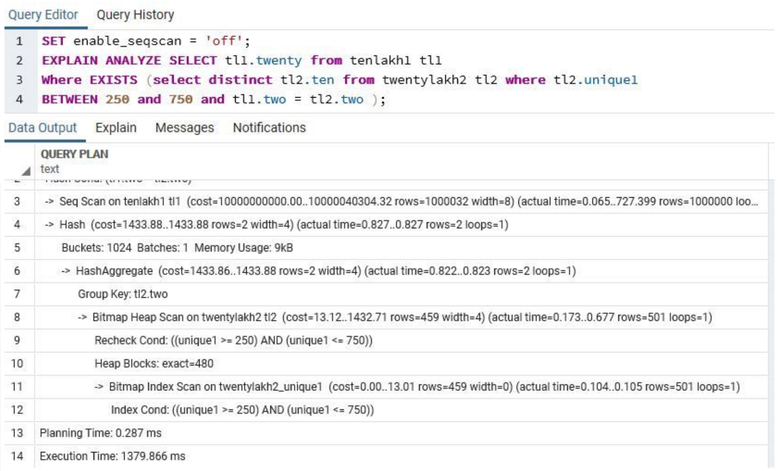
Experiment - 3: Sequential Scan

Results expected & Results found: According to us, if we run the query using the sequential scan, it may in less amount of time. But we think if we toggle it off, the query can take longer time than it does with sequential scan. And the results we found behaved in a similar manner. With the sequential scan turned on we found that the query executed in 1145.6ms. However, with the sequential scan turned off we saw an increase in the query execution time which was 1379.86ms.



```sql
SET enable_seqscan = 'on';
EXPLAIN ANALYZE SELECT tl1.twenty from tenlakh1 tl1
Where EXISTS (select distinct tl2.ten from twentylakh2 tl2 where tl2.unique1
BETWEEN 250 and 750 and tl1.two = tl2.two );
```

```
QUERY PLAN
text
->  Seq Scan on tenlakh1 tl1  (cost=0.00..40304.32 rows=1000032 width=8) (actual time=0.081..625.759 rows=1000000 loops=1)
->  Hash  (cost=1433.88..1433.88 rows=2 width=4) (actual time=1.224..1.224 rows=2 loops=1)
      Buckets: 1024  Batches: 1  Memory Usage: 9kB
      ->  HashAggregate  (cost=1433.86..1433.88 rows=2 width=4) (actual time=1.216..1.217 rows=2 loops=1)
            Group Key: tl2.two
            ->  Bitmap Heap Scan on twentylakh2 tl2  (cost=13.12..1432.71 rows=459 width=4) (actual time=0.224..1.009 rows=501 loops=1)
                  Recheck Cond: ((unique1 >= 250) AND (unique1 <= 750))
                  Heap Blocks: exact=480
                  ->  Bitmap Index Scan on twentylakh2_unique1  (cost=0.00..13.01 rows=459 width=0) (actual time=0.137..0.137 rows=501 loops=1)
                        Index Cond: ((unique1 >= 250) AND (unique1 <= 750))
Planning Time: 0.366 ms
Execution Time: 1145.646 ms
```



```sql
SET enable_seqscan = 'off';
EXPLAIN ANALYZE SELECT tl1.twenty from tenlakh1 tl1
Where EXISTS (select distinct tl2.ten from twentylakh2 tl2 where tl2.unique1
BETWEEN 250 and 750 and tl1.two = tl2.two );
```

```
QUERY PLAN
text
->  Seq Scan on tenlakh1 tl1  (cost=10000000000.00..10000040304.32 rows=1000032 width=8) (actual time=0.065..727.399 rows=1000000 loo...
->  Hash  (cost=1433.88..1433.88 rows=2 width=4) (actual time=0.827..0.827 rows=2 loops=1)
      Buckets: 1024  Batches: 1  Memory Usage: 9kB
      ->  HashAggregate  (cost=1433.86..1433.88 rows=2 width=4) (actual time=0.822..0.823 rows=2 loops=1)
            Group Key: tl2.two
            ->  Bitmap Heap Scan on twentylakh2 tl2  (cost=13.12..1432.71 rows=459 width=4) (actual time=0.173..0.677 rows=501 loops=1)
                  Recheck Cond: ((unique1 >= 250) AND (unique1 <= 750))
                  Heap Blocks: exact=480
                  ->  Bitmap Index Scan on twentylakh2_unique1  (cost=0.00..13.01 rows=459 width=0) (actual time=0.104..0.105 rows=501 loops=1)
                        Index Cond: ((unique1 >= 250) AND (unique1 <= 750))
Planning Time: 0.287 ms
Execution Time: 1379.866 ms
```

Experiment - 4: Bitmap Scan

SQL Query: SELECT tl1.twenty FROM tenlakh1 tl1 WHERE EXISTS (SELECT DISTINCT tl2.ten FROM twentylakh2 tl2 WHERE tl2.unique1 BETWEEN 2499 AND 7499 AND tl2.onepercent BETWEEN 25 AND 75 AND tl1.two = tl2.two);

Parameter: Enables or disables the query planner's use of bitmap-scan plan types. The default is on.A bitmap index scan is a method for joining more than one index. A bitmap index scan works by utilizing the 1st index to find the majority of the rows that fulfill the 1st filter, then utilizing the 2nd filter to find all indexes that fulfill the 2nd filter, after which results are combined to obtain the locations of all rows in the table that fulfill the above two mentioned filters. Later, Postgres will get the rows from the physical table in the order the rows are in the physical table.

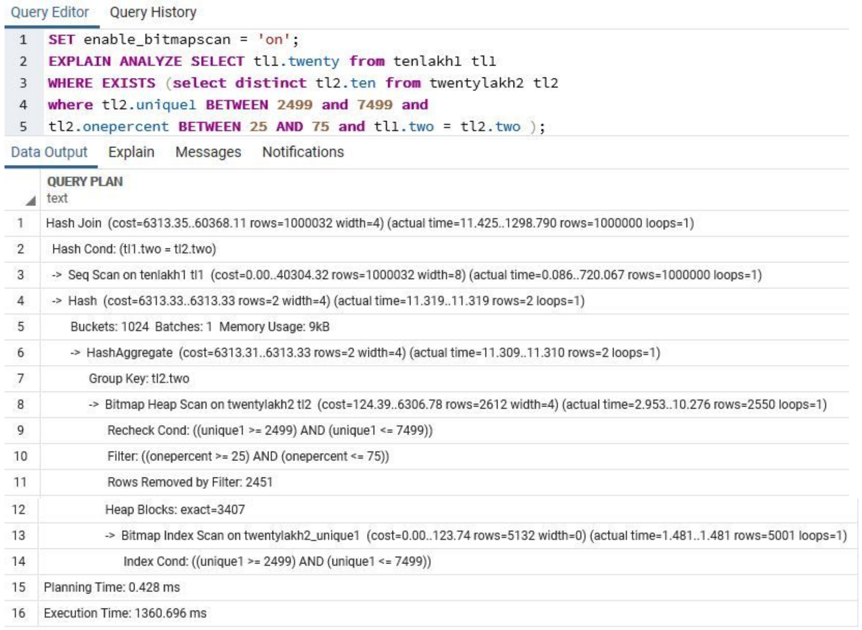
Getting the rows in the physical order from the table that are in the table has it own advantages. In the event that it turns out an enormous part of the table is returned, a bitmap scan turns out to be fundamentally the same as a sequential scan. In specific situations, this can even prompt a bitmap index scan being superior to an ordinary index scan in the event when just a single index is utilized. However, the drawback of this is that Postgres loses knowledge about the sort order.

SET enable_bitmapscan = 'off';

SET enable_bitmapscan = 'on;

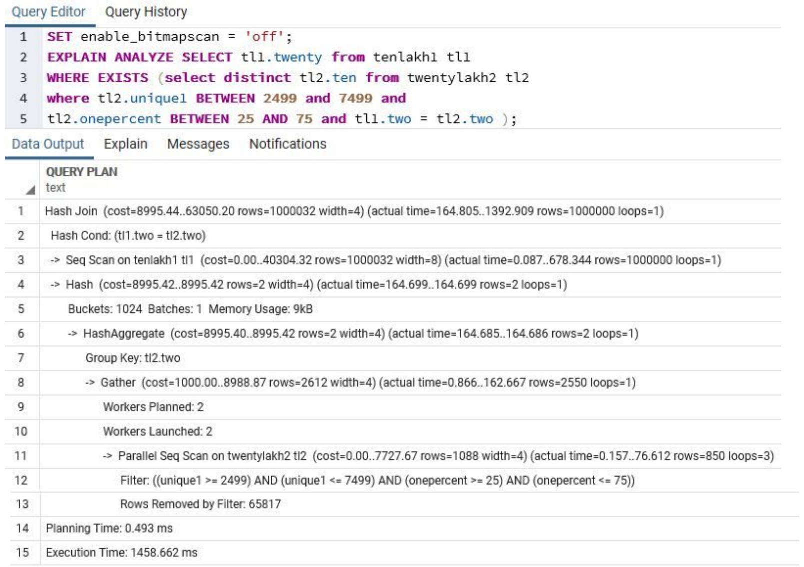
Experiment - 4: Bitmap scan

Results expected & Results found: According to us, the query runs faster when the bitmap scan is being used. When we run the query after turning bitmap scan off, the query takes more time to execute. The query then makes use of the parallel sequential scan. The results we found worked very similar to what we thought. We found the query execution time to be 1360.696ms with bitmap scan turned on and 1458.662ms with bitmap scan turned off.



```sql
SET enable_bitmapscan = 'on';
EXPLAIN ANALYZE SELECT tl1.twenty from tenlakh1 tl1
WHERE EXISTS (select distinct tl2.ten from twentylakh2 tl2
where tl2.unique1 BETWEEN 2499 and 7499 and
tl2.onepercent BETWEEN 25 AND 75 and tl1.two = tl2.two );
```

```
QUERY PLAN
text
Hash Join  (cost=6313.35..60368.11 rows=1000032 width=4) (actual time=11.425..1298.790 rows=1000000 loops=1)
  Hash Cond: (tl1.two = tl2.two)
  ->  Seq Scan on tenlakh1 tl1  (cost=0.00..40304.32 rows=1000032 width=8) (actual time=0.086..720.067 rows=1000000 loops=1)
  ->  Hash  (cost=6313.33..6313.33 rows=2 width=4) (actual time=11.319..11.319 rows=2 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 9kB
        ->  HashAggregate  (cost=6313.31..6313.33 rows=2 width=4) (actual time=11.309..11.310 rows=2 loops=1)
              Group Key: tl2.two
              ->  Bitmap Heap Scan on twentylakh2 tl2  (cost=124.39..6306.78 rows=2612 width=4) (actual time=2.953..10.276 rows=2550 loops=1)
                    Recheck Cond: ((unique1 >= 2499) AND (unique1 <= 7499))
                    Filter: ((onepercent >= 25) AND (onepercent <= 75))
                    Rows Removed by Filter: 2451
                    Heap Blocks: exact=3407
                    ->  Bitmap Index Scan on twentylakh2_unique1  (cost=0.00..123.74 rows=5132 width=0) (actual time=1.481..1.481 rows=5001 loops=1)
                          Index Cond: ((unique1 >= 2499) AND (unique1 <= 7499))
Planning Time: 0.428 ms
Execution Time: 1360.696 ms
```



```sql
SET enable_bitmapscan = 'off';
EXPLAIN ANALYZE SELECT tl1.twenty from tenlakh1 tl1
WHERE EXISTS (select distinct tl2.ten from twentylakh2 tl2
where tl2.unique1 BETWEEN 2499 and 7499 and
tl2.onepercent BETWEEN 25 AND 75 and tl1.two = tl2.two );
```

```
QUERY PLAN
text
Hash Join  (cost=8995.44..63050.20 rows=1000032 width=4) (actual time=164.805..1392.909 rows=1000000 loops=1)
  Hash Cond: (tl1.two = tl2.two)
  ->  Seq Scan on tenlakh1 tl1  (cost=0.00..40304.32 rows=1000032 width=8) (actual time=0.087..678.344 rows=1000000 loops=1)
  ->  Hash  (cost=8995.42..8995.42 rows=2 width=4) (actual time=164.699..164.699 rows=2 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 9kB
        ->  HashAggregate  (cost=8995.40..8995.42 rows=2 width=4) (actual time=164.685..164.686 rows=2 loops=1)
              Group Key: tl2.two
              ->  Gather  (cost=1000.00..8988.87 rows=2612 width=4) (actual time=0.866..162.667 rows=2550 loops=1)
                    Workers Planned: 2
                    Workers Launched: 2
                    ->  Parallel Seq Scan on twentylakh2 tl2  (cost=0.00..7727.67 rows=1088 width=4) (actual time=0.157..76.612 rows=850 loops=3)
                          Filter: ((unique1 >= 2499) AND (unique1 <= 7499) AND (onepercent >= 25) AND (onepercent <= 75))
                          Rows Removed by Filter: 65817
Planning Time: 0.493 ms
Execution Time: 1458.662 ms
```

Conclusion

After performing the above four experiments in PostgreSQL by tweaking and playing around with various configuration parameters, we could conclude that:

1. PostgreSQL comes up with the best query plan based on the number of tuples it retrieves and not execution time alone.
2. With the increase in the size of the working memory, PostgreSQL will execute the queries faster.
3. In Spite of selectivity of a query being more than 10%, we saw that PostgreSQL preferred to use hash joins since it divided the data into multiple batches with multi batch join.
4. The goto option for PostgreSQL for selectivity of a query less than 10% is Index scan. However, if we disable it, it will prefer to use bitmap heap scan which performs slightly better with respect to execution time.
5. Finally, we observed that if the selectivity of a query is greater than 10% then PostgreSQL will prefer to perform the query with sequential scan which performs better than index scan.

Lessons Learned

This project lead us to an expanded comprehension of database framework execution and the effects of database usage on execution. We came across different kinds of configuration parameters that can be used to test the database performance. Mainly, we got an in-depth knowledge of how the runtime of the query is affected just by making a small change in the configuration parameter. The issue that we mainly faced in this part is creating queries that would make us understand the effect of the change in parameters. Ultimately, it gave us a thorough understanding how a wide range of SQL queries can be used to test the performance of a database system.