# CS 487/587 Database Implementation
# Spring 2019
# Database Benchmarking Project - Part III

Amee Sankhesara
Vinaya D Bhat

# Why did we choose Postgres?

- PostgreSQL is a comprehensive, sophisticated database system which offers countless features.
- It is compatible with various platforms using all major languages and middlewares. It also offers most sophisticated locking mechanism. It supports MVCC (Multi version concurrency control) feature.
- PostgreSQL has many configuration parameters in its config file. By tweaking the PostgreSQL config parameters we can improve query performance drastically. So we wanted to learn that how we can get better performance by tweaking the config parameter of PostgreSQL.
- PostgreSQL's query optimizer is superior to many others. We wanted to see the working of optimizer closely and see how it processes different queries.

# Goals

- Learn implementation of join algorithms in PostgresSQL
- Index vs Sequential scan
- Test performance of different types of aggregations
- Memory and execution times

# Implementation

- System Selected: PostgreSQL Version 10.7
- Three tables:

1. Fifteenhundredktup: 1500000 tuples
2. Thousandktup: 1000000 tuples
3. Twothousandktup: 2000000 tuples

# Experiment – 1 : Join Algorithms

**SQL Query :** select * from fifteenhundredktup where unique1 in (select unique1 from thousandktup)

| Expected Results: | Results Obtained: |
| --- | --- |
| • Default: Nested loop join<br>• With enable_nestloop = Off, Postgres chooses Merge Join<br>• With enable_nestloop = Off, enable_mergejoin = Off , Postgres chooses Hash Join | • Default: Hash Join (Average Execution time: 9.9 s)<br>• With enable_hashjoin = Off, Postgres chooses Merge Join (Average Execution time: 19.8 s)<br>• With enable_hashjoin = Off, enable_mergejoin = Off, Postgres chooses Nested Loop Join **(Average Execution time: 7.47 s)** |

**Important observations:**

1. Since selectivity of the query is greater than 10% ,we expected the optimizer to choose Nested Loop Join and last preferable join would be Hash Join since the table is too large to fit in memory. However, Postgres chooses Hash Join over the other joins because of Multi Batch Hash Join. Even though the table size is large, batches are created of **work_mem** size and stored on temporary files on disk.
2. Average execution time of nested loop join is better the other two joins still it is preferred last because Postgres considers the number of tuples the algorithm has to look through. Nested loop join will have to look through 1.5million*1million tuples which is greater than the number of tuples the other algorithms look through.

# Experiment – 1 :

*Varying temp_file_limit on same SQL query*

Temp_file_limit parameter: Specifies the maximum amount of disk space that a process can use for temporary files, such as sort and hash temporary files, or the storage file for a held cursor.

Data Output | Explain | Messages | Notifications

| | QUERY PLAN text |
|---|---|
| 1 | Hash Join (cost=56711.00..212902.70 rows=1000000 width=211) (actual time=1000.836..5331.292 rows=1000000 loops=1) |
| 2 | Hash Cond: (fifteenhundredktup.unique1 = thousandktup.unique1) |
| 3 | -> Seq Scan on fifteenhundredktup (cost=0.00..60455.15 rows=1500015 width=211) (actual time=0.271..1054.891 rows=1500000 loops=1) |
| 4 | -> Hash (cost=40304.00..40304.00 rows=1000000 width=4) (actual time=993.075..993.075 rows=1000000 loops=1) |
| 5 | Buckets: 131072 Batches: 16 Memory Usage: 3227kB |
| 6 | -> Seq Scan on thousandktup (cost=0.00..40304.00 rows=1000000 width=4) (actual time=0.291..647.012 rows=1000000 loops=1) |
| 7 | Planning time: 23.538 ms |
| 8 | Execution time: 5381.021 ms |

Fig(1)
temp_file_limit=-1 (default)
work_mem='4MB' (default)
Enable_hashjoin=on

Postgres can use the total amount of disk space available.

Query Editor | Query History

```
set work_mem='4MB'
set temp_file_limit='1MB'
set enable_hashjoin=on
select * from pg_reload_conf()
explain analyse select * from fifteenhundredktup where unique1 in (select unique1 from thousandktup)
```

Data Output | Explain | Messages | Notifications

```
ERROR:  temporary file size exceeds temp_file_limit (1024kB)
SQL state: 53400
```

Fig(2)
temp_file_limit=1MB
work_mem='4MB' (default)
Enable_hashjoin=on

The query fails since the batch size exceeds the size of temporary file on disk. We expected Postgres to chose then next best join rather than failing. This proves query optimizer does not consider space on disk before choosing the query plan.

If the same query is executed with Enable_hashjoin=off, query will not throw error instead will execute choosing Merge Join.

# Experiment – 1 :

*Varying work_mem on same SQL query*

Work_mem parameter: Specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files.

**Important Observation:**

With decrease in the amount of working memory the number of batches increase since the size of the batch is same as the amount of working memory. So more number of disk access with increase in number of batches.

| | QUERY PLAN<br>text |
|---|---|
| 1 | Hash Join (cost=56711.00..212902.70 rows=1000000 width=211) (actual time=1000.836..5331.292 rows=1000000 loops=1) |
| 2 | Hash Cond: (fifteenhundredktup.unique1 = thousandktup.unique1) |
| 3 | -> Seq Scan on fifteenhundredktup (cost=0.00..60455.15 rows=1500015 width=211) (actual time=0.271..1054.891 rows=1500000 loops=1 |
| 4 | -> Hash (cost=40304.00..40304.00 rows=1000000 width=4) (actual time=993.075..993.075 rows=1000000 loops=1) |
| 5 | Buckets: 131072 Batches: 16 Memory Usage: 3227kB |
| 6 | -> Seq Scan on thousandktup (cost=0.00..40304.00 rows=1000000 width=4) (actual time=0.291..647.012 rows=1000000 loops=1) |
| 7 | Planning time: 23.538 ms |
| 8 | Execution time: 5381.021 ms |

Fig(1)
work_mem='4MB' (default)
Batches=16

| | QUERY PLAN<br>text |
|---|---|
| 1 | Hash Join (cost=56711.00..212902.70 rows=1000000 width=211) (actual time=1069.117..5208.957 rows=1000000 loops=1) |
| 2 | Hash Cond: (fifteenhundredktup.unique1 = thousandktup.unique1) |
| 3 | -> Seq Scan on fifteenhundredktup (cost=0.00..60455.15 rows=1500015 width=211) (actual time=0.064..923.999 rows=1500000 loops=1) |
| 4 | -> Hash (cost=40304.00..40304.00 rows=1000000 width=4) (actual time=1050.712..1050.712 rows=1000000 loops=1) |
| 5 | Buckets: 32768 Batches: 64 Memory Usage: 808kB |
| 6 | -> Seq Scan on thousandktup (cost=0.00..40304.00 rows=1000000 width=4) (actual time=0.046..655.231 rows=1000000 loops=1) |
| 7 | Planning time: 0.425 ms |
| 8 | Execution time: 5249.167 ms |

Fig(2)
work_mem='1MB'
Batches: 64

| | QUERY PLAN<br>text |
|---|---|
| 1 | Hash Join (cost=56711.00..212902.70 rows=1000000 width=211) (actual time=1118.844..5403.024 rows=1000000 loops=1) |
| 2 | Hash Cond: (fifteenhundredktup.unique1 = thousandktup.unique1) |
| 3 | -> Seq Scan on fifteenhundredktup (cost=0.00..60455.15 rows=1500015 width=211) (actual time=0.127..872.352 rows=1500000 loops=1) |
| 4 | -> Hash (cost=40304.00..40304.00 rows=1000000 width=4) (actual time=1050.987..1050.987 rows=1000000 loops=1) |
| 5 | Buckets: 16384 Batches: 128 Memory Usage: 404kB |
| 6 | -> Seq Scan on thousandktup (cost=0.00..40304.00 rows=1000000 width=4) (actual time=0.092..609.749 rows=1000000 loops=1) |
| 7 | Planning time: 0.902 ms |
| 8 | Execution time: 5444.119 ms |

Fig(3)
work_mem='500kB'
Batches: 128

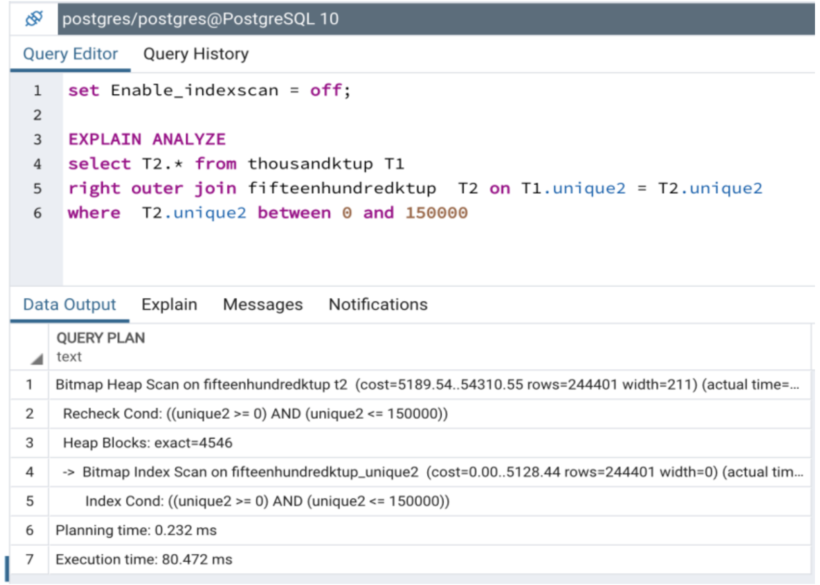
# Experiment-2: Index scan and Sequential scan

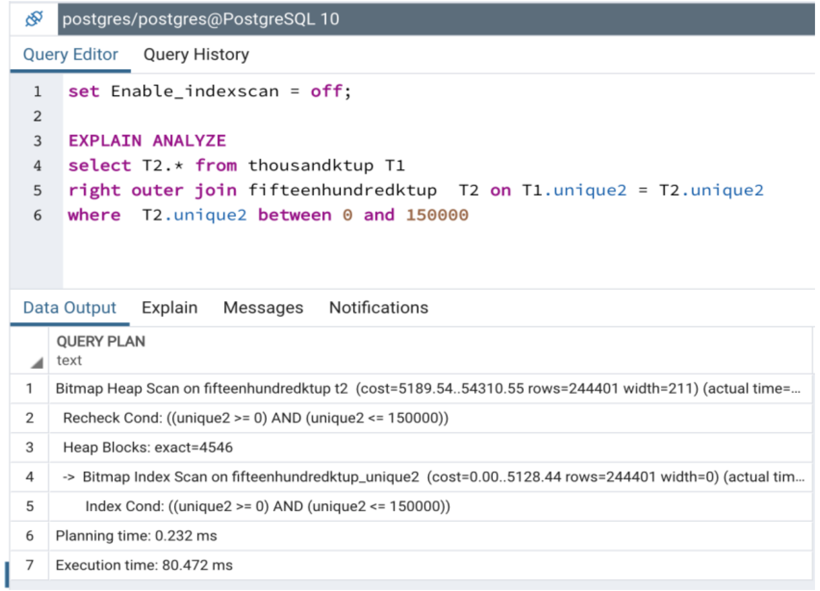
**SQL Query :** select * from fifteenhundredktup where stringu1 like 'AAA%' and unique2 between 0 and 120000

**Expected Result and Results Obtained:** Index Scan (Average Execution Time: 0.797 s ) : Here we are fetching 0 to 120000 number of records from total 1500000 number of records. So, selectivity is less than 10%. Unique2 is primary key of table. So it has clustered index. And selectivity is less than 10% so It will use Index scan for this query according to us. First it will fetch data which has unique2 between 0 and 120000 and then on that data it will apply filter on column stringu1.

**If Enable_indexscan = off**

**Expected Result and Results Obtained:** Bitmap Heap Scan (Average Execution Time : 0.574 s ) : According to us when we set parameter Enable_indexscan to Off it will use Bitmap Heap Scan on table. Because selectivity is less than 10% so for sequential scan there is not enough data. So it will use Bitmap heap scan on both table. First it will find portion of data according to where clause then it will perform join operation on the fly and then it will get result from it.



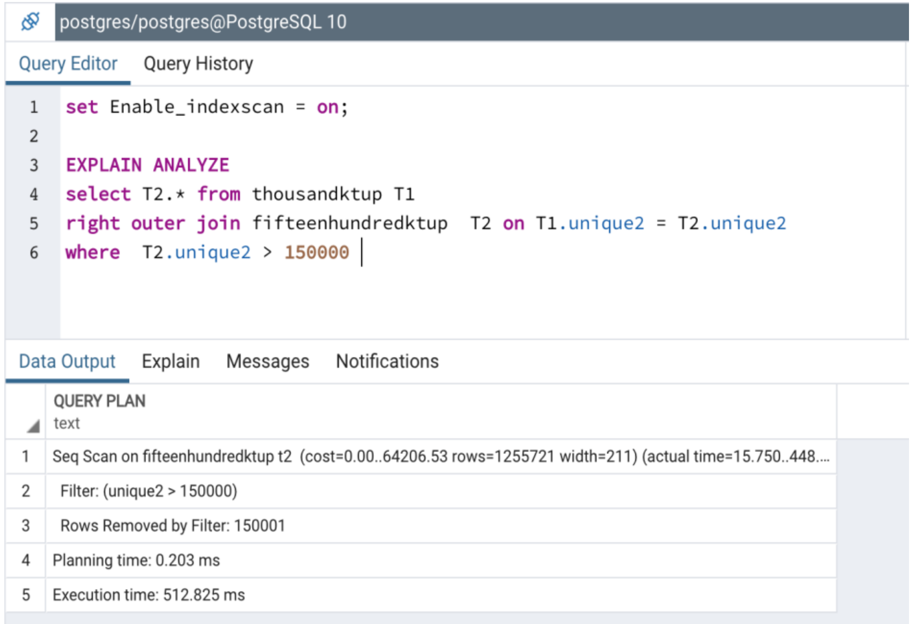
# Experiment-2: Index scan and Sequential scan

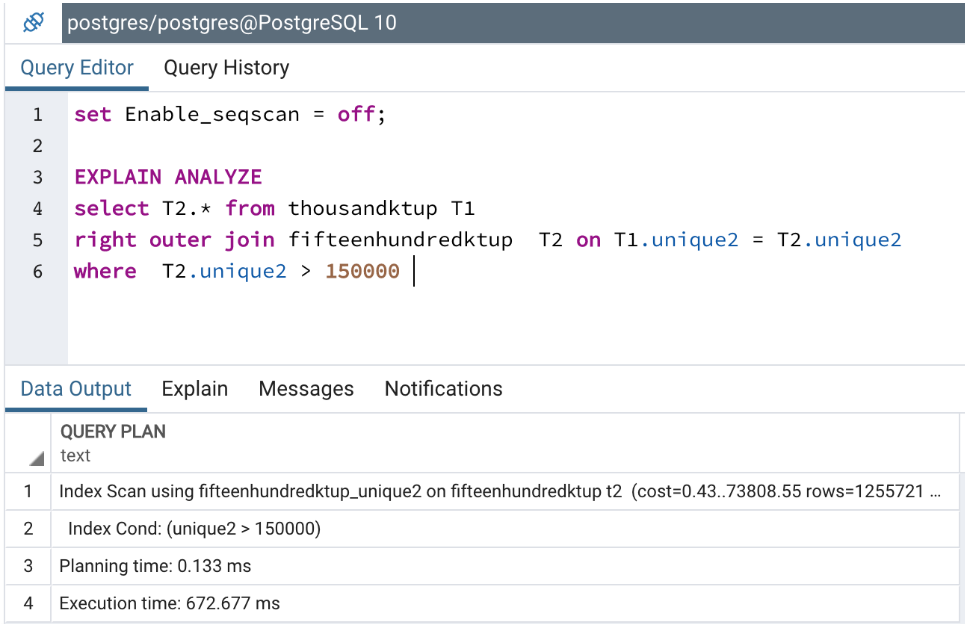
**SQL Query**: select * from fifteenhundredktup where stringu1 like 'AAA%' and unique2 > 150000

**Expected Result and Results Obtained:** Sequential Scan (Average Execution Time: 1.03 s) : Here we are fetching more than 10% records so selectivity is greater than 10%. So it will use sequential scan for this query even though unique2 has clustered index.

If Enable_seqscan = off

**Expected Result and Results Obtained**: Index Scan ( Average Execution Time: 1.238 s) : According to us when we set parameter Enable_seqscan to Off. It will force to use index scan even though its selectivity is greater than 10%. And it will give worse performance than sequential scan.



# Experiment-3: Aggregations

**SQL query**: select oddonepercent,min(evenonepercent) as minevenonepercent from thousandktup group by oddonepercent

Expected Results and Results Obtained were the same. We have first test hash aggregation for that we have just disabled enable_sort parameter and executed above query as hash aggregation do not require sorted input. Then we have test group aggregation for that we have just disabled Enable_hashagg and enabled enable_sort parameter as it requires sorted input. Group aggregation takes more time than hash aggregation as it is performing additional operation sorting operation to sort input.

- enable_sort = off , enable_hashagg = on, Postgres chooses hash aggregate (Average Execution Time: 0.973 s)
- enable_sort = on , Enable_hashagg = off ,Postgres chooses group aggregate (with a sort) (Average Execution Time: 1.77 s)

Data Output Explain Messages Notifications

| | QUERY PLAN text |
|---|---|
| 1 | HashAggregate (cost=10000045305.08..10000045306.08 rows=100 width=8) (actual time=874.401..874.424 rows=100 loops=1) |
| 2 | Group Key: oddonepercent |
| 3 | -> Seq Scan on thousandktup (cost=10000000000.00..10000040304.72 rows=1000072 width=8) (actual time=0.084..519.187 rows=1000000 loops=1) |
| 4 | Planning time: 0.483 ms |
| 5 | Execution time: 874.492 ms |

Data Output Explain Messages Notifications

| | QUERY PLAN text |
|---|---|
| 1 | GroupAggregate (cost=10000167319.26..10000174820.80 rows=100 width=8) (actual time=1643.982..2913.806 rows=100 loops=1) |
| 2 | Group Key: oddonepercent |
| 3 | -> Sort (cost=10000167319.26..10000169819.44 rows=1000072 width=8) (actual time=1641.155..2291.175 rows=1000000 loops=1) |
| 4 | Sort Key: oddonepercent |
| 5 | Sort Method: external merge Disk: 17624kB |
| 6 | -> Seq Scan on thousandktup (cost=10000000000.00..10000040304.72 rows=1000072 width=8) (actual time=0.065..707.731 rows=1000000 loops=1) |
| 7 | Planning time: 0.161 ms |
| 8 | Execution time: 2922.822 ms |

# Experiment-3: Aggregations

- To optimize the query we have created an index on columns present in the aggregation. So not it is doing parallel seq scan on table rather than seq scan. It will be much faster than previous one. It will also give better performance than hash aggregation due to its cache efficiency. HashAggregate needs to keep the whole hash table in memory at once, GroupAggregate only needs the last group. So it will give better performance than previous two cases.
- enable_sort=on, enable_hashagg = off : **group aggregate with index only scan**
- **Average Execution Time: 0.35 s** which is better than previous two cases.

Query Editor    Query History

```
set Enable_sort = on
set Enable_hashagg = off
select * from pg_reload_conf()
create index on thousandktup(oddonepercent,oddonepercent)
explain analyse select oddonepercent,min(evenonepercent) as minevenonepercent
from thousandktup group by oddonepercent

```

Data Output    Explain    Messages    Notifications

| | QUERY PLAN<br>text |
|---|---|
| 1 | Finalize GroupAggregate (cost=80061.50..83212.59 rows=100 width=8) (actual time=244.830..310.789 rows=100 loops=... |
| 2 | Group Key: oddonepercent |
| 3 | -> Gather Merge (cost=80061.50..83210.59 rows=200 width=8) (actual time=244.275..310.755 rows=300 loops=1) |
| 4 | Workers Planned: 2 |
| 5 | Workers Launched: 2 |
| 6 | -> Partial GroupAggregate (cost=79061.48..82187.48 rows=100 width=8) (actual time=215.615..266.984 rows=100 l... |
| 7 | Group Key: oddonepercent |
| 8 | -> Sort (cost=79061.48..80103.15 rows=416667 width=8) (actual time=215.150..239.402 rows=333333 loops=3) |
| 9 | Sort Key: oddonepercent |
| 10 | Sort Method: external merge Disk: 6920kB |
| 11 | -> Parallel Seq Scan on thousandktup (cost=0.00..34470.67 rows=416667 width=8) (actual time=0.103..117.5... |
| 12 | Planning time: 0.362 ms |
| 13 | Execution time: 314.583 ms |

# Experiment-4: Memory management and Execution time using work_mem parameter

**SQL query :** select T2.* from thousandktup T1 inner join fifteenhundredktup T2 on T1.stringu1 = T2.stringu1 and T1.tenpercent = T2.tenpercent inner join twothousandktup T3 on T2.stringu1 = T3.stringu1 and T2.tenpercent = T3.tenpercent

**Output rows:** 1000000 rows

| | QUERY PLAN<br>text |
|---|---|
| 1 | Gather (cost=67047.55..199831.65 rows=10004 width=211) (actual time=4654.255..36379.604 rows=1000000 loops=1) |
| 2 | Workers Planned: 2 |
| 3 | Workers Launched: 2 |
| 4 | -> Nested Loop (cost=66047.55..197831.25 rows=4168 width=211) (actual time=4686.846..35153.161 rows=333333 loops=3) |
| 5 | Join Filter: ((t1.stringu1 = t2.stringu1) AND (t1.tenpercent = t2.tenpercent)) |
| 6 | -> Hash Join (cost=66047.00..168009.53 rows=41670 width=114) (actual time=4685.792..17770.228 rows=333333 loops=3) |
| 7 | Hash Cond: ((t3.stringu1 = t1.stringu1) AND (t3.tenpercent = t1.tenpercent)) |
| 8 | -> Parallel Seq Scan on twothousandktup t3 (cost=0.00..68940.46 rows=833346 width=57) (actual time=0.131..564.325 rows=666667 loops=3) |
| 9 | -> Hash (cost=40304.00..40304.00 rows=1000000 width=57) (actual time=4257.367..4257.367 rows=1000000 loops=3) |
| 10 | Buckets: 1024 (originally 1024) Batches: 2048 (originally 1024) Memory Usage: 93kB |
| 11 | -> Seq Scan on thousandktup t1 (cost=0.00..40304.00 rows=1000000 width=57) (actual time=0.120..663.081 rows=1000000 loops=3) |
| 12 | -> Index Scan using fifteenhundredktup_stringu1 on fifteenhundredktup t2 (cost=0.55..0.70 rows=1 width=211) (actual time=0.051..0.051 rows=1 loops=1000000) |
| 13 | Index Cond: (stringu1 = t3.stringu1) |
| 14 | Filter: (t3.tenpercent = tenpercent) |
| 15 | Planning time: 6.461 ms |
| 16 | Execution time: 36433.934 ms |

# Experiment-4: Memory management and Execution time using work_mem parameter

Important Observations:

With work_mem='100kB' the average execution time is 43.8 s.

With work_mem='4MB' (default) the average execution time is 33.6 s.

With work_mem='128MB' the average execution time is 27.9 s.

**Results Expected and Results Obtained:** We can see that with increase in work_mem the average execution times decreases. Because of work_mem size batches done during hash join with increase in work_mem the number of batches will be less. Also, we can see that two workers are working in parallel in this query to optimize the performance since the number of rows returned is very large.

# Summary/Conclusion

- We have done experiments for query optimization in PostgreSQL by tweaking PostgreSQL config parameters. In the first experiment we have checked different joins. From first experiment we have concluded that even though selectivity of query will be more than 10% still it will use hash join because PostgreSQL is using multi batch join so it will divide data into multiple batches and perform hash join . We have also concluded that PostgreSQL is choosing best query plan based on number of tuples it retrieves not on execution time.
- In the second experiment, we have checked different scans used by query (Index scan, Seq scan, Bitmap heap scan). From second experiment we have concluded that for selectivity less than 10% index scan will give good performance and if we disable config parameter for index scan than it will use bitmap heap scan for selectivity less than 10%. It will also give good performance. And in the case of query having selectivity greater than 10% seq scan is better than index scan.
- In the third experiment, we have checked hash and group aggregation. From the third experiment we have concluded that hash aggregation do not require sorted input so it gives better performance than group aggregation as group aggregation require sorted input so it will take more time than hash aggregation. But group aggregation with index scan only will give better performance than hash aggregation as it will use parallel seq scan on table.
- In the fourth experiment, we have checked memory management and execution time using work_mem parameter. From the fourth experiment, we have concluded that increase in the work_mem will decrease the execution time of query.

# Lessons Learned

Starting this project we were very excited to play around with the Postgres Config parameters. In the course of the project, we learnt that Postgres even though being widely used and popular database still has flaws. Like in the first experiment we were expecting Postgres optimizer to roll over to the next best plan instead of crashing. Another feature of Postgres that took us by surprise was its implementation of Multi Batch Hash Join. Even for tables of size 2 million Postgres still chooses Hash Join because of batches getting created of work_mem size. We learnt a lot of configuration parameter and how they affect the query optimizers choice of query plan. Some of the other parameters we came across and got familiar with are **max_worker_processes, max_parallel_workers_per_gather, max_parallel_workers, max_parallel_workers, seq_page_cost, random_page_cost, min_parallel_table_scan_size** and **min_parallel_index_scan_size.** In the first part of the project we learnt how to populate data from code execution, second part gave us a better understanding about selectivity, indexes and aggregations. Third part of the project increased our understanding of Postgres system. Given more time we would have conducted more number of experiments with different parameters and modification of same query (like with and without subquery, correlated and uncorrelated query etc). After this project we have a better understanding of Postgres, query optimization and queries.