# CS 487/587 Database Implementation
# Winter 2021
# Database Benchmarking Project Part 3

Sara Alotaibi & Sandy Wood

# 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.

# Benchmark Goals

- Modify Postgres configuration parameters to determine how they affect query performance
- Discover what happens at the limits of each configuration parameter - too much memory, too little memory, etc.
- Compare different joins under varying configuration parameters
- Understand how different join algorithms respond to different system parameters.

# Experiment 1: shared_buffers parameter modification

The shared_buffers parameter sets the amount of memory that the system uses for its dedicated buffer memory pool, aside from the memory used for sorting and maintenance operations. Postgres still uses normal operating system cache and memory in addition to this buffer.

| Expected Results: | Results Obtained: |
| --- | --- |
| ● We expect to see the time for the system to execute the query will decrease as memory allocated to the buffers increases.<br>● Past 40%, the performance may start dropping due to overhead of buffer maintenance, less memory available for workers, and less memory available to the operating system's functions and disk cache.<br>● Postgres may not be able to fuw this parameter. | ● As expected, the execution time of the queries decreased with an increase in the buffer size.<br>● Interestingly, the execution time of the query did not decrease at very large memory allocations. We suspect that the operating system's cache may be coming into play somehow.<br>● The Postgres server refused to start with a shared_buffers allocation equal to system memory.<br>● Selectivity did not dramatically change results |

# Experiment 1:

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT * FROM oneKtup
WHERE unique1 BETWEEN 100000 AND 200000

Data Output | Explain | Notifications | Messages

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on onektup (actual rows=100001 loops=1) |
| 2 | Filter: ((unique1 >= 100000) AND (unique1 <= 200000)) |
| 3 | Rows Removed by Filter: 899999 |
| 4 | Planning Time: 0.075 ms |
| 5 | Execution Time: 136.797 ms |

Shared_buffer=’3GB’

Data Output | Explain | Notifications | Messages

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on onektup (actual rows=100001 loops=1) |
| 2 | Filter: ((unique1 >= 100000) AND (unique1 <= 200000)) |
| 3 | Rows Removed by Filter: 899999 |
| 4 | Planning Time: 0.108 ms |
| 5 | Execution Time: 139.101 ms |

Shared_buffer=’2GB’

Data Output | Explain | Notifications | Messages

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on onektup (actual rows=100001 loops=1) |
| 2 | Filter: ((unique1 >= 100000) AND (unique1 <= 200000)) |
| 3 | Rows Removed by Filter: 899999 |
| 4 | Planning Time: 0.250 ms |
| 5 | Execution Time: 183.595 ms |

Shared_buffer=’128MB’

Data Output | Explain | Notifications | Messages

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on onektup (actual rows=100001 loops=1) |
| 2 | Filter: ((unique1 >= 100000) AND (unique1 <= 200000)) |
| 3 | Rows Removed by Filter: 899999 |
| 4 | Planning Time: 0.336 ms |
| 5 | Execution Time: 391.993 ms |

Shared_buffer=’16MB’

# Experiment 1:

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT * FROM oneKtup
WHERE unique1 BETWEEN 100000 AND 110000

Data Output | Explain | Notifications | Messages

| | QUERY PLAN text |
|---|---|
| 1 | Gather (actual rows=10001 loops=1) |
| 2 | Workers Planned: 2 |
| 3 | Workers Launched: 2 |
| 4 | -> Parallel Seq Scan on onektup (actual rows=3334 loops=3) |
| 5 | Filter: ((unique1 >= 100000) AND (unique1 <= 110000)) |
| 6 | Rows Removed by Filter: 330000 |
| 7 | Planning Time: 0.084 ms |
| 8 | Execution Time: 135.113 ms |

Shared_buffer=’3GB’

# Experiment 2: Join algorithm:

**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<br>Postgres chooses Nested Loop Join<br>(Average Execution time: 7.47 s) |

# Experiment 2: Modifying parameters on same join - temp_file_limit

| Expected Results: temp_file_limit | Results Obtained: |
| --- | --- |
| ● Creating a temp_file_limit that exceeds the available temporary file on the disk will cause Postgres to have a performance decrease.<br>● Worse join algorithm or an increased use of OS disk space. | ● If enable_hashjoin=on:<br>○ Postgres raised an error and did not complete the query if batch size exceeded temp disk space.<br>○ Planner has to choose a failing hash, hands are tied<br>● If enable_hashjoin=off:<br>○ Postgres chose a merge join instead |

# Experiment 2:

*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

```
1 set work_mem='4MB'
2 set temp_file_limit='1MB'
3 set enable_hashjoin=on
4 select * from pg_reload_conf()
5 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 3: work_mem

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

| Expected Results: work_mem | Results Obtained: |
| --- | --- |
| ● We expect performance to improve significantly as work_mem is increased, up until a certain point. At this point, the entire relation will fit into memory and the hash join will not need any further memory.<br>● If the memory required to contain the entire hash table is never provided, we will not see an optimal performance of the hash join | ● Disk reads increased as expected with each decrease of work_mem<br>● Execution time was fairly consistent - we suspect this is due to the OS cache being used rather than a true disk read.<br>○ Need to look into exactly what's happening in the OS |

# Experiment 3: work_mem

*Varying work_mem on same SQL query*

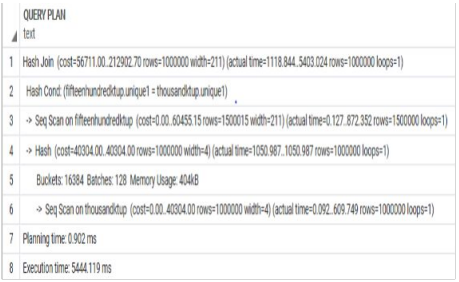
Work_mem parameter: Specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files.

**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.


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


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




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

# Experiment 4: Change work_mem on aggregate queries

**SQL Query :** INSERT INTO TMP SELECT SUM (ONEMTUP1.unique3) FROM ONEMTUP1 GROUP BY ONEMTUP1.onePercent

| Expected Results: | Results Obtained: |
| --- | --- |
| ● We expect that as the amount of work memory increases, there will be an increase in the performance of the aggregate query as more of the hash table that is used to perform the aggregate query appears simultaneously in memory.<br>● At a certain point, the benefit will plateau, as the maximum work memory provided will exceed the work memory needed to complete the aggregate query | ● As the work_mem dedicated to aggregate queries grew, the execution time of the queries decreased. This was true for work memory sizes up to 500MB.<br>● At work mem sizes of 1GB+, the execution time of the queries plateaued, and actually slightly increased by fractions of a millisecond. We suspect this is due to the overhead of memory management. |

# Experiment 4: Change work_mem on aggregate queries

Set work_mem='4MB';
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT SUM (oneKtup.unique3) FROM oneKtup
GROUP BY oneKtup.one_percent

Data Output | Explain | Notifications | Messages

| | QUERY PLAN text |
|---|---|
| 1 | Finalize GroupAggregate (actual rows=20 loops=1) |
| 2 | Group Key: one_percent |
| 3 | -> Gather Merge (actual rows=60 loops=1) |
| 4 | Workers Planned: 2 |
| 5 | Workers Launched: 2 |
| 6 | -> Sort (actual rows=20 loops=3) |
| 7 | Sort Key: one_percent |
| 8 | Sort Method: quicksort Memory: 25kB |
| 9 | Worker 0: Sort Method: quicksort Memory: 25kB |
| 10 | Worker 1: Sort Method: quicksort Memory: 25kB |
| 11 | -> Partial HashAggregate (actual rows=20 loops=3) |
| 12 | Group Key: one_percent |
| 13 | -> Parallel Seq Scan on onektup (actual rows=333333 loops=3) |
| 14 | Planning Time: 14.075 ms |
| 15 | Execution Time: 3294.888 ms |

work_mem='4MB'

Data Output | Explain | Notifications | Messages

| | QUERY PLAN text |
|---|---|
| 1 | Finalize GroupAggregate (actual rows=20 loops=1) |
| 2 | Group Key: one_percent |
| 3 | -> Gather Merge (actual rows=60 loops=1) |
| 4 | Workers Planned: 2 |
| 5 | Workers Launched: 2 |
| 6 | -> Sort (actual rows=20 loops=3) |
| 7 | Sort Key: one_percent |
| 8 | Sort Method: quicksort Memory: 25kB |
| 9 | Worker 0: Sort Method: quicksort Memory: 25kB |
| 10 | Worker 1: Sort Method: quicksort Memory: 25kB |
| 11 | -> Partial HashAggregate (actual rows=20 loops=3) |
| 12 | Group Key: one_percent |
| 13 | -> Parallel Seq Scan on onektup (actual rows=333333 loops=3) |
| 14 | Planning Time: 0.080 ms |
| 15 | Execution Time: 744.567 ms |

work_mem='500MB'

# Experiment 4: Change work_mem on aggregate queries

Set work_mem='4MB';
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT SUM (oneKtup.unique3) FROM oneKtup
GROUP BY oneKtup.one_percent

Data Output | Explain | Notifications | Messages

| | QUERY PLAN text |
|---|---|
| 1 | Finalize Read-only column tual rows=20 loops=1) |
| 2 | Group Key: one_percent |
| 3 | -> Gather Merge (actual rows=60 loops=1) |
| 4 | Workers Planned: 2 |
| 5 | Workers Launched: 2 |
| 6 | -> Sort (actual rows=20 loops=3) |
| 7 | Sort Key: one_percent |
| 8 | Sort Method: quicksort Memory: 25kB |
| 9 | Worker 0: Sort Method: quicksort Memory: 25kB |
| 10 | Worker 1: Sort Method: quicksort Memory: 25kB |
| 11 | -> Partial HashAggregate (actual rows=20 loops=3) |
| 12 | Group Key: one_percent |
| 13 | -> Parallel Seq Scan on onektup (actual rows=333333 loops=3) |
| 14 | Planning Time: 0.101 ms |
| 15 | Execution Time: 752.861 ms |

work_mem='1GB'

Data Output | Explain | Notifications | Messages

| | QUERY PLAN text |
|---|---|
| 1 | Finalize GroupAggregate (actual rows=20 loops=1) |
| 2 | Group Key: one_percent |
| 3 | -> Gather Merge (actual rows=60 loops=1) |
| 4 | Workers Planned: 2 |
| 5 | Workers Launched: 2 |
| 6 | -> Sort (actual rows=20 loops=3) |
| 7 | Sort Key: one_percent |
| 8 | Sort Method: quicksort Memory: 25kB |
| 9 | Worker 0: Sort Method: quicksort Memory: 25kB |
| 10 | Worker 1: Sort Method: quicksort Memory: 25kB |
| 11 | -> Partial HashAggregate (actual rows=20 loops=3) |
| 12 | Group Key: one_percent |
| 13 | -> Parallel Seq Scan on onektup (actual rows=333333 loops=3) |
| 14 | Planning Time: 0.075 ms |
| 15 | Execution Time: 773.939 ms |

work_mem='2GB'

# Conclusions

- Postgres can fail - if you make its job hard
  - Postgres was really good at planning queries given strange circumstances.
  - Only when we made the parameter settings strange and limited its options did it fail to execute a query.
  - Postgres will not start under certain parameter settings.

- More memory does not necessarily translate to faster execution
  - Depending on available system resources, query type, and size of the data, speedup may plateau or even decrease as more memory is allocated to processes.

- Operating Systems are good at being fast
  - When it comes to caching disk pages, the operating system is highly efficient and used to doing it regularly. Finding conditions where it will perform poorly is a difficult task.
  - Modern systems are much more complicated than one OS for one computer with one disk.

# Lessons Learned

- Many Postgres parameters, particularly relating to costs, are influenced by many factors and have complex interactions.
- Parameters can be affected by each other as well as system components. There is no perfect setting or paradigm that will always produce optimal results.
- Performance is highly dependant on query structure and size.
- The operating system plays a larger part in disk/memory management than was expected.

# Questions?

# Thank you