# PROJECT PART III: BENCHMARK IMPLEMENTATION

## CS 487/587 DATABASE IMPLEMENTATION SPRING 2019


PROJECT MEMBERS:

AJINKYA SHINDE

CHUNWEI LI

# Choice of System

We choose option 2 and evaluate a single system - Postgres to do benchmark because Postgres system provides users with more options and interfaces to adjust system performance. It better helps understand what & how parameters impact on database performances. Originally we chose BigQuery in part 1, but we found BigQuery doesn't provide enough documents about how to change system parameters to adjust system performance, so we decide to switch to Postgres in order to better focus on database performance and understand implementation of database systems.

# Summary of the Approach to the Benchmark

## Four experiments

- 10% rule performance

---

- Aggregates performance
- Effect of buffer cache size on join
- Effect of work_mem on hash join

## Paramters

- enable_indexscan
- work_mem
- effective_cache_size
- enable_hashjoin
- shared_buffers

# Experiment 1 - 10% Rule Performance

1. This experiment explores when it is good to use an unclustered index vs. not using an index vs. using a clustered index
2. Use a 100,000 tuple relation (scaled up version of TENKTUP2)
3. Use Wisconsin Bench queries 2, 4 and 6. Run queries 2, 4, and 6 on the same dataset. Query 2 ran without an index on unique2, Query 4 ran with a clustered index on unique2, Query 6 requires an unclustered index on unique1.
4. When using index, parameter enable_indexscan of PostgreSQL was set to on, and parameter enable_seqscan was set to off.
5. Metric is elapsed time.
6. Expect using a cluster index has the best performance, an unclustered index has second best performance, and not using an index has the worst performance.

# Experiment 1 - result

| Query 2 without index | Query 4 with clustered index on unique2 | Query 6 with unclustered index on unique1 |
|---|---|---|
| mydb=# EXPLAIN ANALYZE INSERT INTO TMP SELECT * FROM TENKTUP2 WHERE unique2 BETWEEN 792 AND 10791;<br>QUERY PLAN<br>---<br>Insert on tmp (cost=0.00..4531.00 rows=10048 width=211) (actual time=89.465..89.465 rows=0 loops=1)<br>-> Seq Scan on tenktup2 (cost=0.00..4531.00 rows=10048 width=211) (actual time=0.789..50.691 rows=10000 loops=1)<br>Filter: ((unique2 >= 792) AND (unique2 <= 10791))<br>Rows Removed by Filter: 90000<br>Planning Time: 0.474 ms<br>Execution Time: 93.580 ms | mydb=# EXPLAIN ANALYZE INSERT INTO TMP SELECT * FROM TENKTUP2 WHERE unique2 BETWEEN 792 AND 10791;<br>Insert on tmp (cost=0.29..621.25 rows=10048 width=211) (actual time=19.370..19.370 rows=0 loops=1)<br>-> Index Scan using unique2_idx on tenktup2 (cost=0.29..621.25 rows=10048 width=211) (actual time=0.031..3.938 rows=10000 loops=1)<br>Index Cond: ((unique2 >= 792) AND (unique2 <= 10791))<br>Planning Time: 0.386 ms<br>Execution Time: 19.399 ms | mydb=# EXPLAIN ANALYZE INSERT INTO TMP SELECT * FROM TENKTUP2 WHERE unique1 BETWEEN 792 AND 10791;<br>Insert on tmp (cost=214.94..4517.92 rows=10014 width=211) (actual time=115.545..115.545 rows=0 loops=1)<br>-> Bitmap Heap Scan on tenktup2 (cost=214.94..4517.92 rows=10014 width=211) (actual time=26.341..37.311 rows=10000 loops=1)<br>Recheck Cond: ((unique1 >= 792) AND (unique1 <= 10791))<br>Rows Removed by Index Recheck: 60452<br>Heap Blocks: exact=858 lossy=2070<br>-> Bitmap Index Scan on unique1_idx (cost=0.00..212.43 rows=10014 width=0) (actual time=26.173..26.173 rows=10000 loops=1)<br>Index Cond: ((unique1 >= 792) AND (unique1 <= 10791))<br>Planning Time: 0.139 ms<br>Execution Time: 115.585 ms |
| mydb=# INSERT INTO TMP SELECT * FROM TENKTUP2 WHERE unique2 BETWEEN 792 AND 10791;<br>INSERT 0 10000<br>Time: 60.756 ms | mydb=# INSERT INTO TMP SELECT * FROM TENKTUP2 WHERE unique2 BETWEEN 792 AND 10791;<br>INSERT 0 10000<br>Time: 41.717 ms | mydb=# INSERT INTO TMP SELECT * FROM TENKTUP2 WHERE unique1 BETWEEN 792 AND 10791;<br>INSERT 0 10000<br>Time: 51.658 ms |

The above results are what were expected. Using a cluster index has the best performance, an unclustered index has the second best performance, and not using an index has the worst performance.

# Experiment 2 - Aggregates Performance

1. This experiment explores what parameters improve aggregate performance.
2. Use a 100,000 tuple relation (scaled up version of TENKTUP2).
3. Use Wisconsin Bench queries 20 and 23. Run queries 20 and 23 on the same dataset.
4. Run queries with different parameters - work_mem, temp_buffers, seq_page_cost, effective_cache_size.
5. Metric is elapsed time.
6. Expect work_mem impacts on performance, but temp_buffers, seq_page_cost, and effective_cache_size don't have effective impact on performance.

# Experiment 2 - result

| work_mem | 64KB | 80KB | 96KB | 112KB | 128KB | Performance becomes better when increasing work_mem, but when using index for query 23, index has more impact on performance than work_mem does, so increase of work_mem does not improve performance. |
|---|---|---|---|---|---|---|
| Query 20 | 23.185 ms | 12.884 ms | 12.803 ms | 12.47 ms | 12.715 ms | |
| Query 23 | 0.795 ms | 0.828 ms | 0.806 ms | 0.781 ms | 0.855 ms | |

| temp_buffers | 800KB | 1600KB | 2400KB | 3200KB | 4000KB | As expected, temp_buffers does not impact on performance of aggregates query because temp_buffers is used by temporary tables, and aggregate query does not have large temporary tables. |
|---|---|---|---|---|---|---|
| Query 20 | 13.119 ms | 13.4 ms | 14.102 ms | 13.686 ms | 12.646 ms | |
| Query 23 | 0.928 ms | 0.815 ms | 0.782 ms | 0.820 ms | 0.790 ms | |

| seq_page_cost | 1 | 2 | 3 | 4 | 5 | Result shows that increase of seq_page_cost slightly improves performance of aggregate query. |
|---|---|---|---|---|---|---|
| Query 20 | 16.875 ms | 13.023 ms | 12.640 ms | 12.522 ms | 11.747 | |
| Query 23 | 0.819 ms | 0.770 ms | 0.855 ms | 0.728 ms | 0.834 ms | |

| effective_cache_size | 8KB | 16KB | 80KB | 160KB | 320KB | Result shows that increase of effective_cache_size slightly improves performance of aggregate query. |
|---|---|---|---|---|---|---|
| Query 20 | 14.304ms | 14.869 | 14.170ms | 12.632 ms | 11.607 ms | |
| Query 23 | 1.008 ms | 0.852 ms | 0.826 ms | 0.711 ms | 0.735 ms | |

Through this experiment, we learned that both seq_page_cost and effective_cache_size can slightly improve performance of aggregate query against what is expected.

# EXPERIMENT 3: EFFECT OF BUFFER CACHE SIZE

- **Specification :** Query plan evaluation by change in buffer cache size
- Change the '*shared_buffers*' configuration parameter in *postgresql.conf*
- Tells how much memory is allocated to PostgreSQL for caching the data.
- Technically, how much size for buffer pool space to be allocated
- Firstly, understand the data flow in PostgreSQL to understand the parameter significance
- Pre – experiment run precautions : To avoid corruption of results and to measure actual execution time

1. Change the '*shared_buffers'* in *postgresql.conf*
2. *Clear the OS and PG Shared cache* using utility RamMap
3. *Run the benchmark queries*

**Evaluation of benchmark test suite – Important metrics**

- **First-run execution time – Check the query execution time across test suite results when the benchmark queries are run for the first time.**
- **Shared hits / Shared-Read – Check how many hits/misses happen at buffer cache across the test suite results**
- **Final execution time - Check the execution time after cache size comes into effect on 2nd run**

**TEST SUITES**

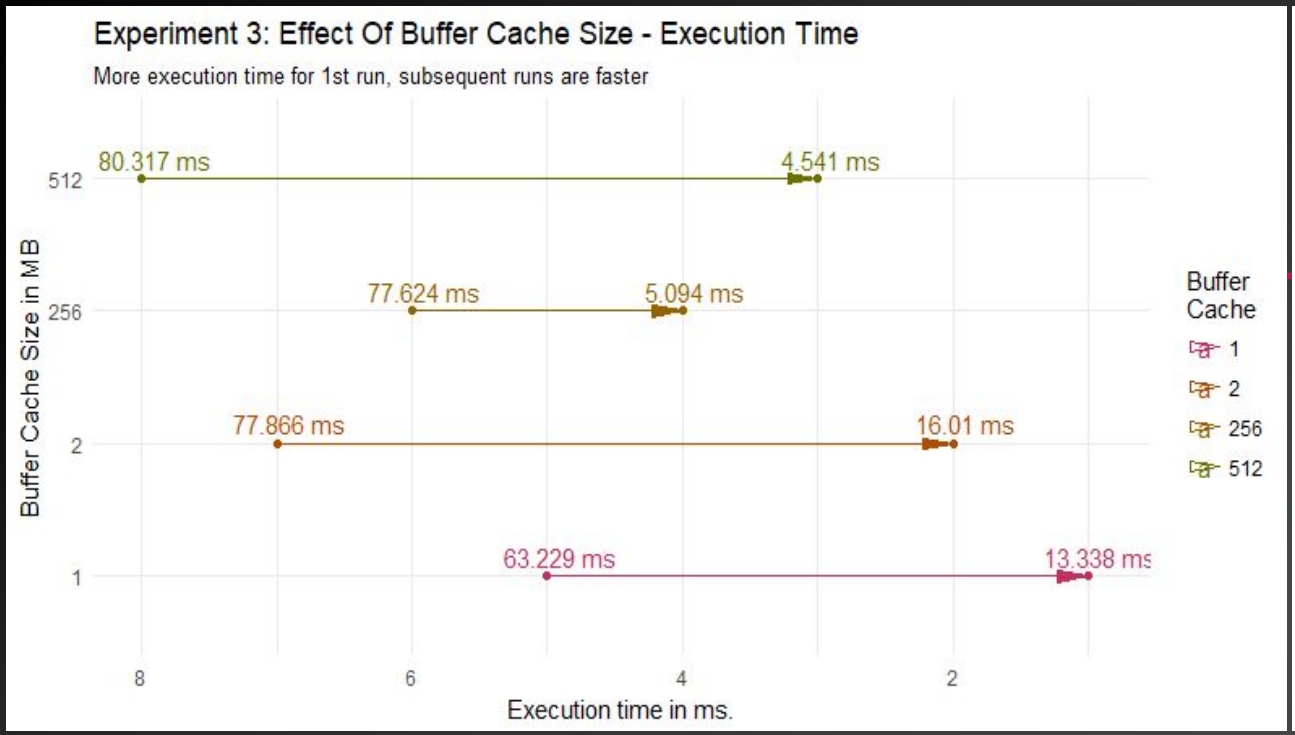
- **Test Suite I – Join Query with selectivity**

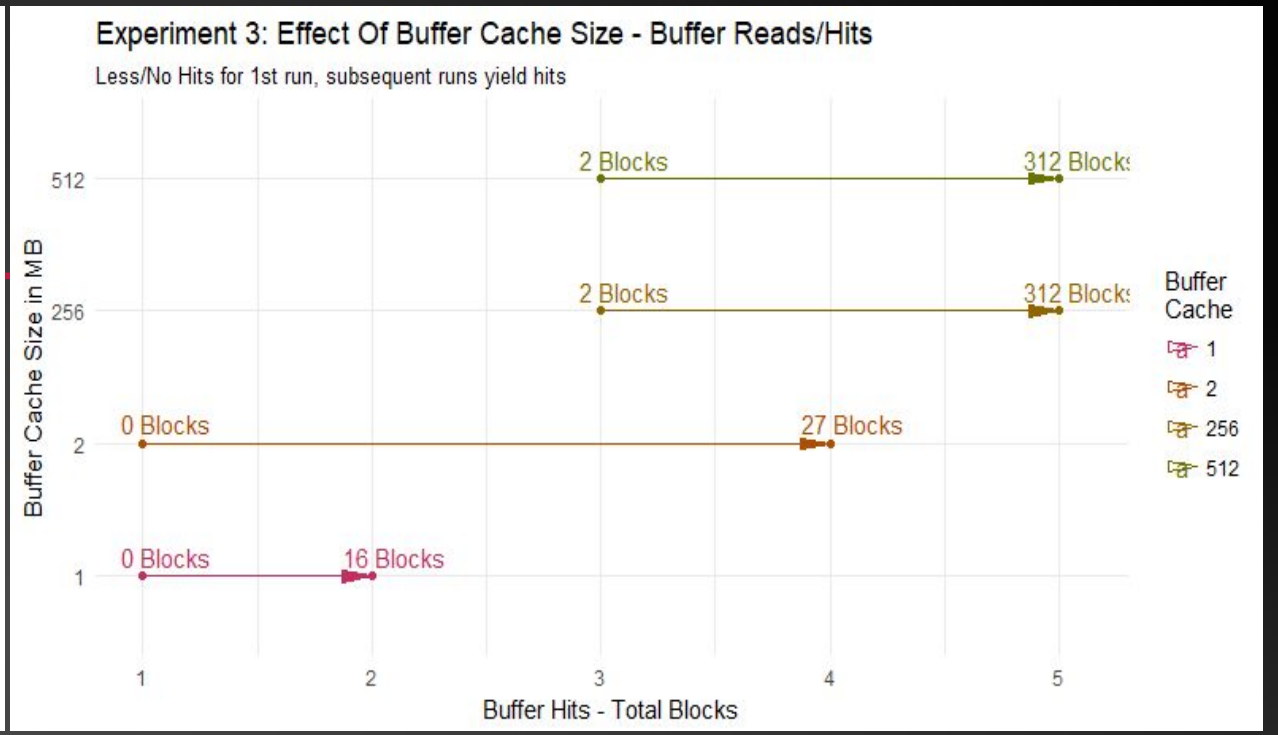
```
EXPLAIN (ANALYZE,BUFFERS) SELECT A.*,B.* FROM TENKTUP1 AS A JOIN TENKTUP2 AS B ON A.UNIQUE2 = B.UNIQUE2 WHERE unique2 BETWEEN 792 AND 1791;
```

- **Test Suite II – Select Query with Index**

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM TENKTUP1 WHERE unique2 BETWEEN 792 AND 1791;
```

# EXPERIMENT 3: EFFECT OF BUFFER CACHE SIZE - RESULTS




## SAMPLE OUTPUT – TEST SUITE II RUN

- 512 MB Cache Size - 1st run

```
wisconsinbenchmk=# EXPLAIN (ANALYZE,BUFFERS) SELECT A.*,B.*  FROM TENKTUP1 AS A JOIN TENKTUP2 AS B ON  A.UNIQUE2 = B.UNIQUE2 WHERE A.unique2 B
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=79.75..518.01 rows=999 width=432) (actual time=45.813..76.965 rows=1000 loops=1)
   Hash Cond: (b.unique2 = a.unique2)
   Buffers: shared hit=6 read=345
   ->  Seq Scan on tenktup2 b  (cost=0.00..412.00 rows=10000 width=216) (actual time=0.251..43.732 rows=10000 loops=1)
         Buffers: shared hit=2 read=310
   ->  Hash  (cost=67.27..67.27 rows=999 width=216) (actual time=25.812..25.812 rows=1000 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 252kB
         Buffers: shared hit=1 read=35
         ->  Index Scan using tenktup1_pkey on tenktup1 a  (cost=0.29..67.27 rows=999 width=216) (actual time=1.418..21.929 rows=1000 loops=1)
               Index Cond: ((unique2 >= 792) AND (unique2 <= 1791))
               Buffers: shared hit=1 read=35
 Planning time: 411.057 ms
 Execution time: 80.317 ms
(13 rows)
```

**Expected – More the buffer size**

- **Less execution timeObserved - More the buffer size**
- **Higher execution time on 1st run**
- **More time to load the buffer cache pool**
- **Subsequent runs faster**

# EXPERIMENT 4 : EFFECT OF work_mem ON HASH JOIN

1. work_mem allows to define the working memory available for query operations.
2. Initially, the experiment involved changing the hash table size. But, the only way to do this is by changing the work_mem parameter. Hence, the experiment has been renamed as above.
3. If the operation in the query takes memory larger than the value specified in the work_mem , then postgresql performs operation on the data that has been spilled on to the disk
4. To set work_mem execute SET work_mem='1MB'; in psql (current session).
5. To reset back to default , execute RESET work_mem; in psql (current session).

# EXPERIMENT 4 - RESULT

**Hash Join with Sort (EXPLAIN (ANALYZE,BUFFERS) SELECT A.*,B.* FROM TENKTUP1 AS A JOIN TENKTUP2 AS B ON A.UNIQUE1 = B.UNIQUE1 WHERE A.unique1 BETWEEN 792 AND 1791 AND B.unique1 BETWEEN 792 AND 1791 ORDER BY A.stringu1;)**

work_mem = 64KB

```
QUERY PLAN
------------------------------------------------------------------------
-
 Sort  (cost=1064.56..1064.81 rows=100 width=432)
(actual time=63.286..67.859 rows=1000 loops=1)
   Sort Key: a.stringu1
   Sort Method: external merge  Disk: 456kB
   Buffers: shared hit=229 read=395, temp read=164
written=167
   ->  Hash Join
         Hash Cond: (a.unique1 = b.unique1)
         Buffers: shared hit=229 read=395, temp read=64
written=56
         ->  Seq Scan on tenktup1 a
               Filter: ((unique1 >= 792) AND (unique1 <=
1791))
               Rows Removed by Filter: 9000
               Buffers: shared hit=98 read=214
         ->  Hash
               Buckets: 256  (originally 256)   Batches: 8
(originally 4)  Memory Usage: 63kB
               Buffers: shared hit=131 read=181, temp
written=21
               ->  Seq Scan on tenktup2 b
 Planning time: 0.581 ms
 Execution time: 71.225 ms
(20 rows)
```

work_mem = 500 KB

```
QUERY PLAN
------------------------------------------------------------------------
 Sort  (cost=944.56..944.81 rows=100 width=432)
(actual time=45.533..46.849 rows=1000 loops=1)
   Sort Key: a.stringu1
   Sort Method: external merge  Disk: 440kB
   Buffers: shared hit=286 read=338, temp read=55
written=55
   ->  Hash Join
         Hash Cond: (a.unique1 = b.unique1)
         Buffers: shared hit=286 read=338
         ->  Seq Scan on tenktup1 a
               Filter: ((unique1 >= 792) AND (unique1 <=
1791))
               Rows Removed by Filter: 9000
               Buffers: shared hit=133 read=179
         ->  Hash
               Buckets: 1024  Batches: 1  Memory Usage:
252kB
               Buffers: shared hit=153 read=159
               ->  Seq Scan on tenktup2 b
                     Filter: ((unique1 >= 792) AND (unique1
<= 1791))
                     Rows Removed by Filter: 9000
                     Buffers: shared hit=153 read=159
 Planning time: 0.648 ms
 Execution time: 49.184 ms
(20 rows)
```

work_mem = 1MB

```
QUERY PLAN
------------------------------------------------------------------------
 Sort  (cost=944.56..944.81 rows=100 width=432)
(actual time=45.871..46.009 rows=1000 loops=1)
   Sort Key: a.stringu1
   Sort Method: quicksort  Memory: 540kB
   Buffers: shared hit=284 read=340
   ->  Hash Join
         Hash Cond: (a.unique1 = b.unique1)
         Buffers: shared hit=284 read=340
         ->  Seq Scan on tenktup1 a
               Filter: ((unique1 >= 792) AND (unique1 <=
1791))
               Rows Removed by Filter: 9000
               Buffers: shared hit=136 read=176
         ->  Hash
               Buckets: 1024  Batches: 1  Memory Usage:
252kB
               Buffers: shared hit=148 read=164
               ->  Seq Scan on tenktup2 b
 Planning time: 0.642 ms
 Execution time: 47.037 ms
(20 rows)
```

The above results where as per what we expected to be. Since the work_mem is initially set to 64kb (default), sorting is done on disk storage. However, as work_mem is increased to 1MB i.e. almost the 2X size used previously 440kB, the sorting is done in memory

# Conclusion

- Thus, depending on the query characteristics , the different parameters need to be adjusted accordingly to gain the necessary throughput and less overhead overall
- As seen from the above experiments, we think mainly the characterestics of the query need to be analysed for proper benchmarking
- In general, we need to think about indices, selectivity and different operations like sorting, join, distinct clauses to perform proper benchmarking and achieve meaningful results.

# Lessons Learned

- Setting parameter and measuring query performance is an incremental process
- Thus, there is no perfect value for a parameter in different postgresql environments. Depending on the database environment setup, the parameter values will differ across different systems.