Wisconsin Benchmark

CS 487-587 Database Implementations

Minjin Enkhjargal, Yan Li

PostgreSQL

- Option 2 to analyze a single system - PostgreSQL
- Previous experience
- Free and open-source software
- User-friendly interface
- Resourceful, reliable, fast performance
- Provides easy optimizer tuning options

Goals

- Learn how to design a benchmark.
- Learn how different factors such as parameters, query operators, query types, and relation sizes can affect the performance using PostgreSQL.
- Learn how to tune the PostgreSQL optimizer.
- Carry out experiments to analyze performance.

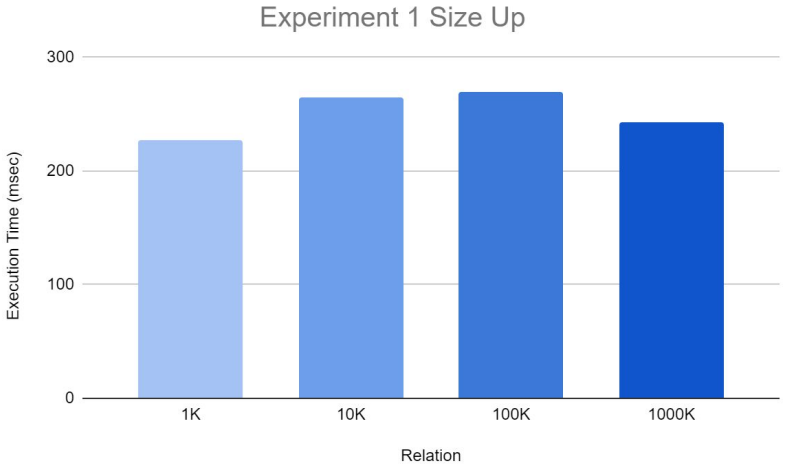
Experiment 1

1. **Sizeup**
    a. Experiment Specifications
        i. **Purpose:** Explores how the system responds to different sizes of clustered relations
        ii. **Data:** Use 1k, 10k, 100k, 1000k relations.
        iii. **Queries Used:** Use query 7 and run the query on the different size relations, compare how query execution time changes as the database size changes.
        iv. **Parameters:** No parameters changed in this test.
        v. **Expected Results:** Query 7 uses a clustered index and is looking for when unique2 = 2001. Since it is looking for one tuple and uses an ordered index, we expect execution to be fairly quick. We expect performance to become slower for larger relations, but not substantially.

Experiment 1 Results

Query 7 - single tuple selection via clustered index to screen
SELECT * FROM HUNDREDKTUP
WHERE unique2 = 2001

Relation	Trial 1 (ms)	Trial 2 (ms)	Trial 3 (ms)	Trial 4 (ms)	Average (ms)
1K	236	242	195	236	227.25
10K	278	259	264	257	264.50
100K	313	235	273	258	269.75
1000K	196	272	261	245	243.50



Experiment 1 (cont.) Different Results?

Query 7 - single tuple selection via clustered index to screen
SELECT * FROM HUNDREDKTUP
WHERE unique2 = 2001

Refresh and mix in different queries in between trials

Relation	Trial 1 (ms)	Trial 2 (ms)	Trial 3 (ms)	Trial 4 (ms)	Trial 5 (ms)	Trial 6 (ms)	Average (ms)
1K	358	418	328	350	373	381	368
10K	278	259	264	284	291	289	277.5
100K	313	235	273	236	355	292	284
1000K	209	258	355	923	735	826	626

Discussion

- The actual results is not quite as expected. The size of the relation actually does not affect the query execution time significantly when the data is clustered. It does not perform slower as the size increases in this case due to the clustered index.
- Observation: When the same query is executed consecutively, it is faster and more consistent, when compare to the trials with refreshing the server and mixing in other queries in between.

Experiment 2

1. **Test 2 queries with and without a hash aggregate plan type**
    a. Experiment Specifications
        i. **Purpose:** Explores how queries are affected when the query planner either uses or does not the hash aggregate plan type
        ii. **Data:** Use 100K tuple relations.
        iii. **Query Used:** Use Wisconsin Benchmark queries Query 20 (no index) and Query 23(with clustered index). Run each query with the enable_hashagg turned on and off.
        iv. **Parameters:** enable_hashagg (boolean).
        v. **Expected Results:** The expected result is that when the enable_hashagg is turned on, these queries will likely perform faster, since they use an aggregate operator. On the other hand, when the enable_hashagg is turned off, the queries will likely run slower. When this parameter is disabled, it will likely default to a merge sort on disk, which will cause the execution to be slower.

Experiment 2

Query Tuning in Postgresql.conf file

```
#------------------------------------------------
# QUERY TUNING
#------------------------------------------------

# - Planner Method Configuration -

#enable_bitmapscan = on
#enable_hashagg = on
#enable_hashjoin = on
#enable_indexscan = on
#enable_indexonlyscan = on
#enable_material = on
#enable_mergejoin = on
#enable_nestloop = on
#enable_parallel_append = on
#enable_seqscan = on
#enable_sort = on
#enable_incremental_sort = on
#enable_tidscan = on
#enable_partitionwise_join = off
#enable_partitionwise_aggregate = off
#enable_parallel_hash = on
#enable_partition_pruning = on
```

```
#------------------------------------------------
# QUERY TUNING
#------------------------------------------------

# - Planner Method Configuration -

#enable_bitmapscan = on
#enable_hashagg = off
#enable_hashjoin = on
#enable_indexscan = on
#enable_indexonlyscan = on
#enable_material = on
#enable_mergejoin = on
#enable_nestloop = on
#enable_parallel_append = on
#enable_seqscan = on
#enable_sort = on
#enable_incremental_sort = on
#enable_tidscan = on
#enable_partitionwise_join = off
#enable_partitionwise_aggregate = off
#enable_parallel_hash = on
#enable_partition_pruning = on
```

Experiment 2 Results

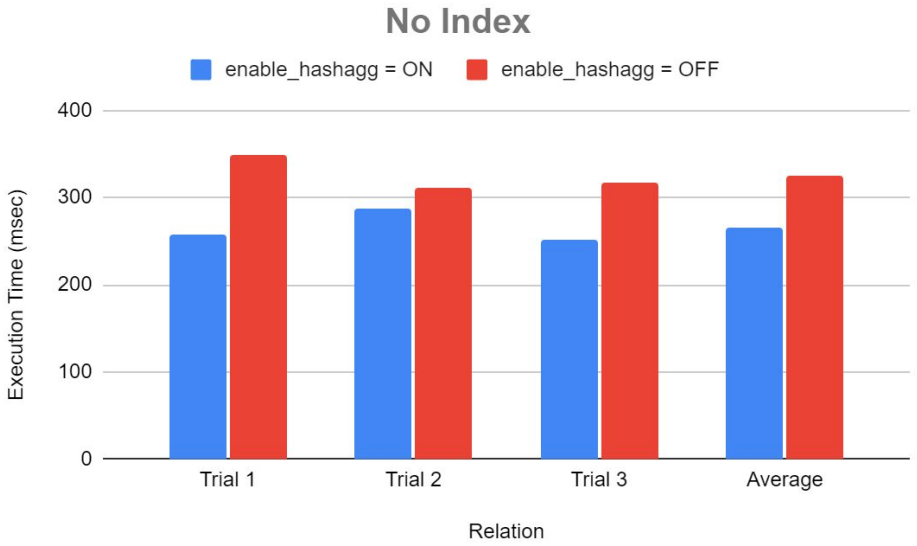
Query 20 (no index)

Minimum Aggregate Function

INSERT INTO hundredktup_ni_tmp1

SELECT min(hundredktup_noindex.twenty) FROM hundredktup_noindex

Relation	enable_hashagg ON (msec)	enable_hashagg OFF (msec)
Trial 1	257	348
Trial 2	287	311
Trial 3	251	317
Average	265	325.33

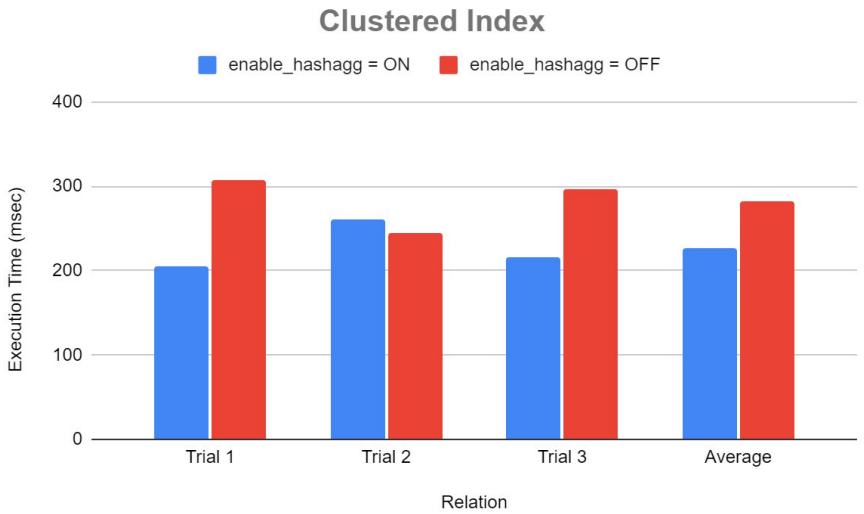


Experiment 2(cont.)

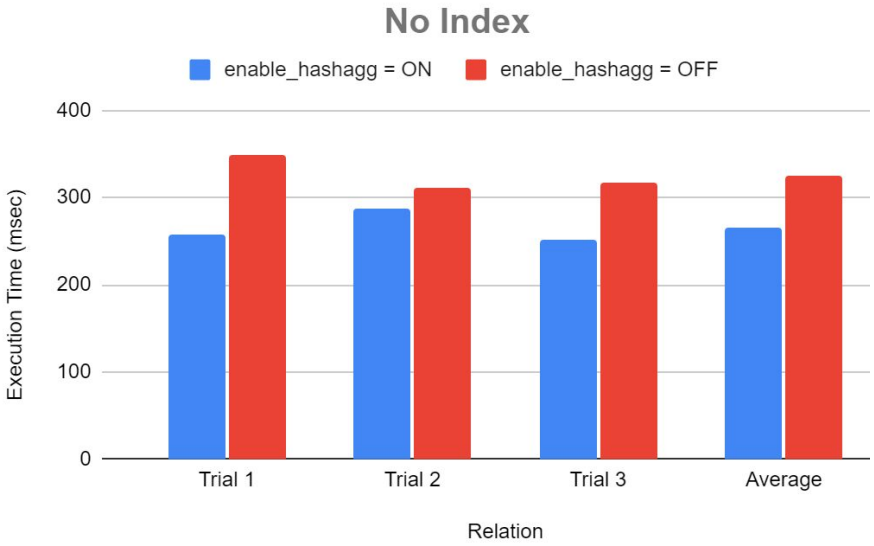
Query 23 (with clustered index)
Minimum Aggregate Function
INSERT INTO hundredktup_ni_tmp1
SELECT min(hundredktup_noindex.twenty) FROM hundredktup_noindex

Clustered Index

Relation	enable_hashagg ON (msec)	enable_hashagg OFF (msec)
Trial 1	205	308
Trial 2	260	245
Trial 3	215	296
Average	226.67	283



No Index VS Clustered Index with enable_hashagg



Discussion

- The actual results match with our expected results.
- Overall, the query executes faster when the parameter enable_hashagg is turned on in the clustered index relation.
- Since query 20 uses an aggregate operator, the query will likely perform faster when the enable_hashagg is turned on. When enable_hashagg is turned off, the sort aggregate become default, therefore, performs slower.

Experiment 3

1. Test the 3 join algorithms used by PostgreSQL.
    a. Experiment specifications
        i. **Purpose:** This experiment will explore how the 3 join algorithms perform for the same query.
        ii. **Data:** Use 100K and 1000K tuple relations.
        iii. **Queries Used:** Use Wisconsin Bench query 13 (clustered index). We will run the query 3 times. For each round, we disable 2 parameters while enabling 1.
        iv. **Parameters:** *enable_nestloop, enable_hashjoin and enable_mergejoin*. These parameters are BOOL, and can be enabled and disabled.
        v. **Expected Results:** Query 13 uses a simple join and uses a clustered index. Since this query will use the 1000K tuple, BPRIME will contain 10% of the data, which is 100K. Since the two tables 100K and 1000K are different and larger in size, the hash-join algorithm should run the fastest. Merge join should come in second place in terms of performance, because it performs well executing a clustered index and larger data sets. The reason it'd be slower than hash join is that the tables vary too much in their sizes. Nested loop joins perform well when one table is significantly small and uses indices, since it requires fewer comparisons. In our experiment, nested loop joins will likely perform the worst, since the data sizes may be too large for this type of join algorithm to perform well. The second table BPRIME contains 10% of the first table (100K), which we do not believe is significantly smaller.

Experiment 3 Results

NEST LOOP
SET enable_hashjoin = off;
SET enable_mergejoin = off;
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT * FROM thousandktup, hundredktup
WHERE (thousandktup.unique2 = hundredktup.unique2)

HASH JOIN
SET enable_mergejoin = off;
SET enable_nestloop = off;
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT * FROM thousandktup, hundredktup
WHERE (thousandktup.unique2 = hundredktup.unique2)

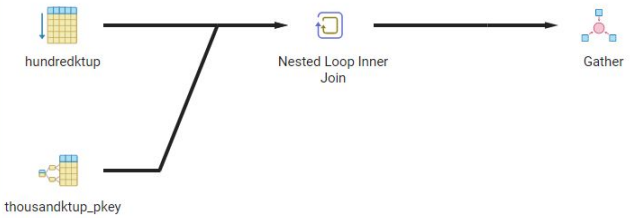
MERGE JOIN
SET enable_hashjoin = off;
SET enable_nestloop = off;
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT * FROM hundredktup, tenkup1
WHERE (hundredktup.unique2 = tenkup1.unique2)

Type of Join	Trial 1 (sec)	Trial 2 (sec)	Trial 3 (sec)	Average (sec)
Nest loop	0.378	0.415	0.433	0.409
Hash join	3.837	2.146	2.037	2.673
Merge join	1.484	1.549	1.360	1.465

Discussion: Nested Loop

Nested Loop: Scans the outer relation sequentially. For each result row, scans the inner relation for matching rows.

- Sequential scan is done on 100K, index is not really needed here
- Index scan is done on 1000K table. Having index on the join key of the inner relation helps.



postgres/postgres@PostgreSQL 13

Query Editor | Query History

```
1  SET enable_hashjoin = off;
2  SET enable_mergejoin = off;
3  EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
4  SELECT * FROM thousandktup, hundredktup
5  WHERE (thousandktup.unique2 = hundredktup.unique2)
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (actual rows=100000 loops=1)
2	-> Seq Scan on hundredktup (actual rows=100000 loops=1)
3	-> Index Scan using thousandktup_pkey on thousandktup (actual rows=1 loops...
4	Index Cond: (unique2 = hundredktup.unique2)
5	Planning Time: 0.625 ms
6	Execution Time: 415.208 ms

Discussion: Merge Join

Merge Join: Records in the outer and inner relation are compared until there is a possibility of join clause matching. (Sort both relations and merge rows.) Only used if there is at least one join condition with the = operator. Scans through only once.

- Is good when there is an index on the join keys of both relations
- Usually good when both tables are larger and similar size
- Our tables (100K vs 1000K) were different sized

Query Editor | Query History

```
1  SET enable_hashjoin = off;
2  SET enable_nestloop = off;
3  EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
4  SELECT * FROM thousandktup, hundredktup
5  WHERE (thousandktup.unique2 = hundredktup.unique2)
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Merge Join (actual rows=100000 loops=1)
2	Merge Cond: (thousandktup.unique2 = hundredktup.unique2)
3	-> Index Scan using thousandktup_pkey on thousandktup (a...
4	-> Index Scan using hundredktup_pkey on hundredktup (actu...
5	Planning Time: 2.305 ms
6	Execution Time: 1483.548 ms

Discussion: Hash Join

Hash Join: A Hash table is built using the inner relation. Hash key is calculated based on the join clause. Outer relation is hashed using the join clause key to find matching entries in the hash table.

- When the hash table doesn't fit in memory, partition both sides of the join into some number of batches.
- Partitioning Required: used “parallel hash join” instead of hash join
- Batches: 8, Memory Usage: 3200kB (3.2MB)
- Cost: 3 * (# pages in B + # pages in O)
- In ideal scenario for hash join, the cost is: (# pages) + (# pages)

Query Editor | Query History

```
1  SET enable_nestloop = off;
2  SET enable_mergejoin = off;
3  EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
4  SELECT * FROM thousandktup, hundredktup
5  WHERE (thousandktup.unique2 = hundredktup.unique2)
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Gather (actual rows=100000 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Hash Join (actual rows=33333 loops=3)
5	Hash Cond: (thousandktup.unique2 = hundredktup.unique2)
6	-> Parallel Seq Scan on thousandktup (actual rows=333333 loops=3)
7	-> Parallel Hash (actual rows=33333 loops=3)
8	Buckets: 16384 Batches: 8 Memory Usage: 3200kB
9	-> Parallel Seq Scan on hundredktup (actual rows=33333 loops=3)
10	Planning Time: 8.191 ms
11	Execution Time: 3836.583 ms

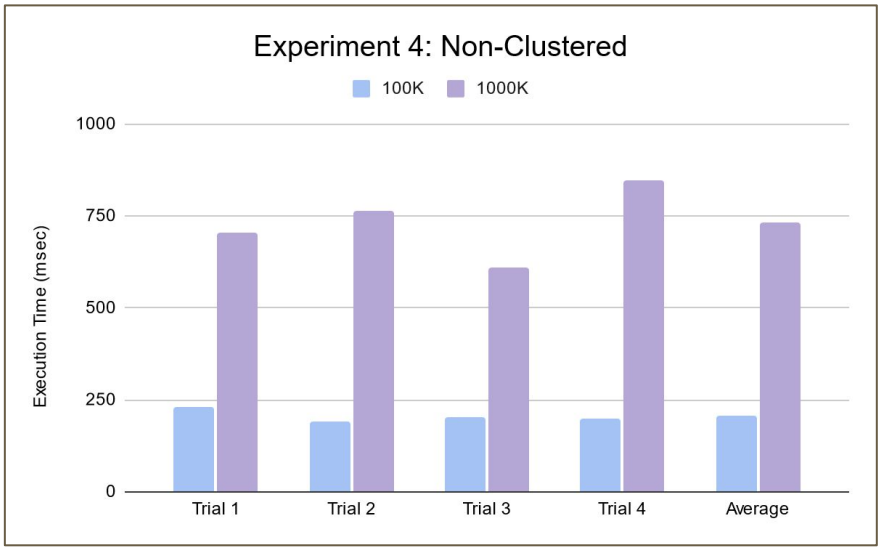
Experiment 4

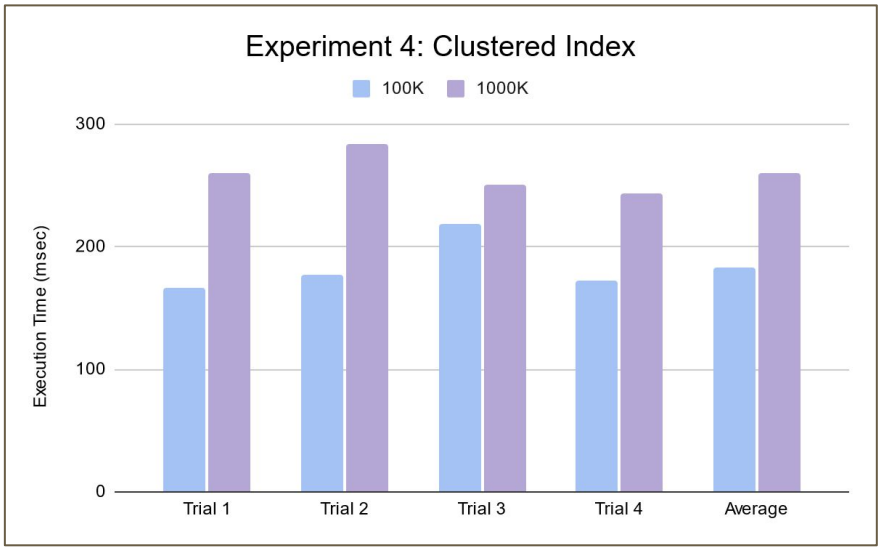
1. Test clustered vs nonclustered of two different sized tuple relations
    a. Experiment Specifications
        i. **Purpose:** Explores how two different sizes of relations would respond differently to clustered vs nonclustered indices.
        ii. **Data:** Use 100K and 1000K tuple relations.
        iii. **Query Used:** Use Wisconsin Benchmark queries 7(clustered index), 15(non-clustered index). Run these queries on the 100k and 1000k tuple relations. Compare each query on the different sized tuples to see their performances.
        iv. **Parameters:** No parameters changed in this test
        v. **Expected Results:** The expected result is that the clustered index tuple relation will be faster than the non-clustered index for the 100K tuple relation. Since this is a larger data set, having a clustered index to determine the order of the data may allow the execution

Experiment 4 Results

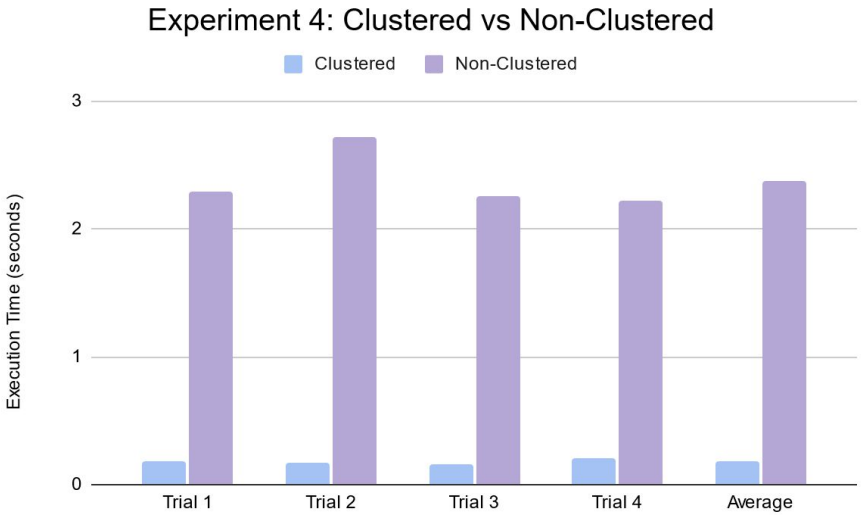
SELECT * FROM thousandktup
WHERE unique2 = 2001

SELECT * FROM hundredktup
WHERE unique2 = 2001





Experiment 4 Results



Query 15 - JoinAselB

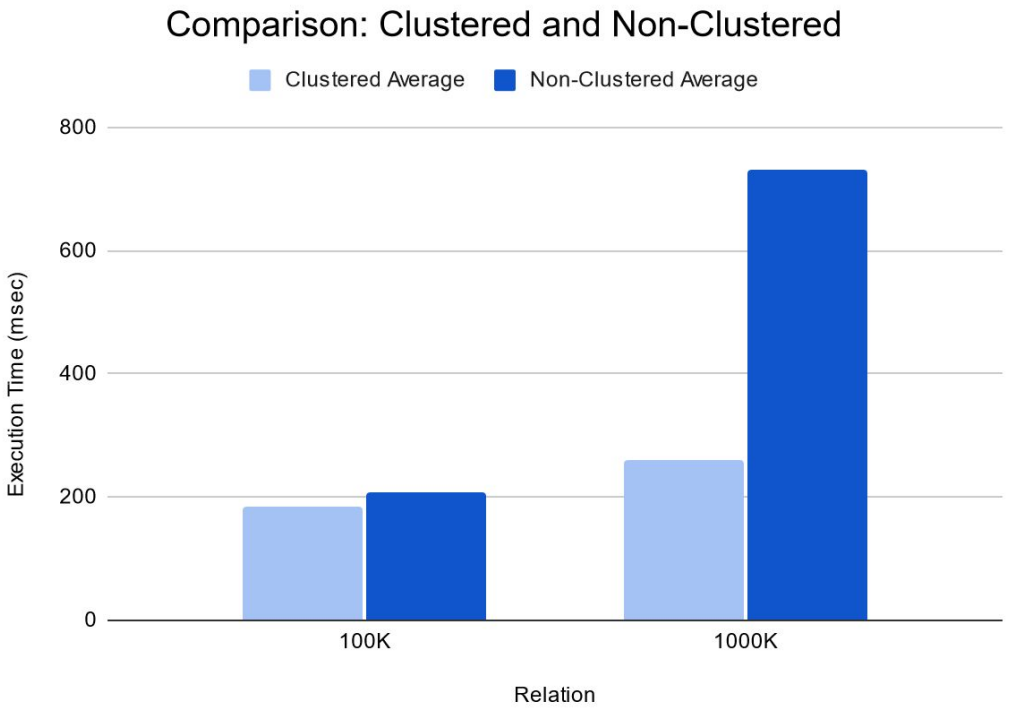
```
SELECT * FROM thousandktup, hundredktup
WHERE (thousandktup.unique1 = hundredktup.unique1)
AND (thousandktup.unique2 < 1000)
```

Query 15 - JoinAselB

```
SELECT * FROM  thousandktup_nonc,  hundredktupk_nonc
WHERE ( thousandktup_nonc.unique1 =  hundredktupk_nonc.unique1)
AND ( thousandktup_nonc.unique2 < 1000)
```

Discussion

- Sequential access: In this type of index, a search can go straight to a specific point in data so that you can keep reading sequentially from there.
- Clustered index method uses location mechanism to locate index entry at the start of a range.
- Clustering on an index forces the physical ordering of the data to be the same as the index order of the index chosen.



Conclusion

- The query execution time does not change significantly in correspondence to increase in relation size, when the relation is clustered.
- The query executes faster when enable_hashagg is turned on in comparison to off, regardless of the type of relation (No index vs clustered).
- If the optimizer underestimates a row count, it may choose a nested loop join and scan the inner relation more often than expected. If the optimizer overestimates a row count, it may choose hash or merge join which may require a scan of both relations completely, worse performance than a nested loop join with an index on the inner relation. When a hash table does not fit into memory, performance takes a hit.
- Using clustered indexes showed better performance than non-clustered

Lessons Learned

- Wisconsin Benchmark
- Learned how to tune PostgreSQL query optimizer
- Running the same query multiple times might show reduced execution time. It may be important to refresh the shared buffer to refresh cached data
- When a hash table is too large, it is split up into multiple batches the size of work_mem and hash join is run on each batch (parallel hash join). We thought it would be the fastest for experiment 3!
- In the queries we ran, the clustered index did help with performance - we may see a more substantial result if we ran queries that use ranged neighboring data (pulling date range... etc.)
- CLUSTER in PostgreSQL is not updated automatically; May need to re-cluster

References

- https://www.enterprisedb.com/postgres-tutorials/parallel-hash-joins-postgresql-explained
- http://patshaughnessy.net/2016/1/22/is-your-postgres-query-starved-for-memory
- https://www.postgresql.org/
- https://www.guru99.com/clustered-vs-non-clustered-index.html
- https://severalnines.com/database-blog/overview-join-methods-postgresql
- https://www.cybertec-postgresql.com/en/join-strategies-and-performance-in-postgresql/
- https://drill.apache.org/docs/sort-based-and-hash-based-memory-constrained-operators/
- https://deepai.org/publication/sort-based-grouping-and-aggregation
- Inside the PostgreSQL Shared Buffer Cache (2ndquadrant.com)
- https://www.postgresonline.com/journal/archives/10-How-does-CLUSTER-ON-improve-index-performance.html

Questions?