PostgreSQL Explorations

Andrew Stevenson and Pragina Vaidya

GCP VM Specifications

Series: E2-medium

RAM: 4 GB

OS: Ubuntu 16.04LTS

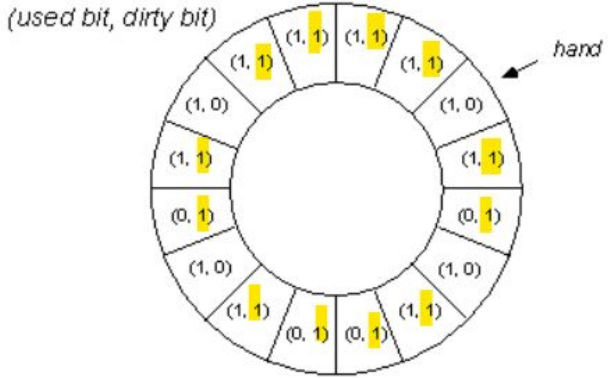
Database: PostgreSQL (PostgreSQL 11.10 (Debian 11.10-0+deb10u1)

- Free and open-source RDBMS

Benchmark Design Motivations

- To better understand the PostgreSQL buffer manager and query optimizer
- To fully implement the Wisconsin Benchmark table design
- To create experiments of general interest to the psql community
- To increase mastery of PostgreSQL configuration options and buffer cache behavior

Experiment 1 Motivation & Setup



- If psql is running the unmodified **clock algorithm**, it should be possible to fill the buffer with **dirty pages**.
- These dirty pages will then need to be written to disk before other pages can be loaded.

<ins>Central Question</ins>: **Can we impact the performance of subsequent read-only queries, by writing to a large number of tuples?**

<ins>Expectation 1</ins>: If psql runs the clock algorithm with little/no modification, then 'dirty' queries executing immediately after a large write query will see a **negative performance delta** when compared to 'clean' queries having no preceding write.

<ins>Expectation 2</ins>: If psql runs the clock algorithm with little/no modification, we expect:

- a small delta when buffer is smaller than relation size,
- a larger delta when buffer is same as relation size,
- and no delta when the buffer is more than 2x the relation size.

Image from: https://people.engr.ncsu.edu/efg/501/f98/lectures/annotations/a23a.html

Experiment 1 Results


```sql
--Control Group
SELECT * FROM miltup2;
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM miltup1;
```

```sql
--Experimental Group
UPDATE miltup2 SET string4 = 'data overwritten exp1 round 1';
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM miltup1;
```

RESULTS:

- A small but consistent delta in performance at 128 MB and 208 MB
- Delta peaks when relation size is same as buffer size
- The 500 MB data are suspect, as VM performance became unstable at this point

Raw Data (readings in ms)

Experimental Rounds	128 MB	208 MB	500 MB
1	224.556	250.607	232.203
2	216.405	223.227	314.094
3	215.797	280.902	239.712
4	218.992	217.27	242.79
5	215.309	240.614	1614.352
Avg	**218.2118**	**242.524**	**528.6302**
Control Group Rounds			
1	209.086	238.194	247.879
2	211.182	216.778	858.793
3	208.441	229.014	226.985
4	206.114	213.95	849.303
5	219.718	214.068	1311.452
Avg	**210.9082**	**222.4008**	**698.8824**

Experiment 1 Note: Source files...

From bufmgr.c

```
 * MarkBufferDirty() -- mark a pinned buffer's contents as "dirty".
 *      The disk write is delayed until buffer replacement or checkpoint.
 *
```

From freelist.c

```
/*
 * ClockSweepTick - Helper routine for StrategyGetBuffer()
 *
 * Move the clock hand one buffer ahead of its current position and return the
 * id of the buffer now under the hand.
 */
static inline uint32
ClockSweepTick(void)
{
```

https://github.com/postgres/postgres/blob/39b03690b529935a3c33024ee68f08e2d347cf4f/src/backend/storage/buffer/

Experiment 2

- Explores postgres system performance when query result size is greater than configured working memory
- Compares hash aggregation performance between indexed and non-indexed attributes
    - with inadequate working memory
    - running with aggregate and group by
- Expected Results:
    - Reduced memory size will generate inefficient hash tables for larger relations like *hundredktup* with size greater than working memory.
    - When working memory is held equal, an aggregate or *group by* query on an indexed attribute having 100% selection **will be slower**, when compared to a query on a *non-indexed* attribute having 20% selection.

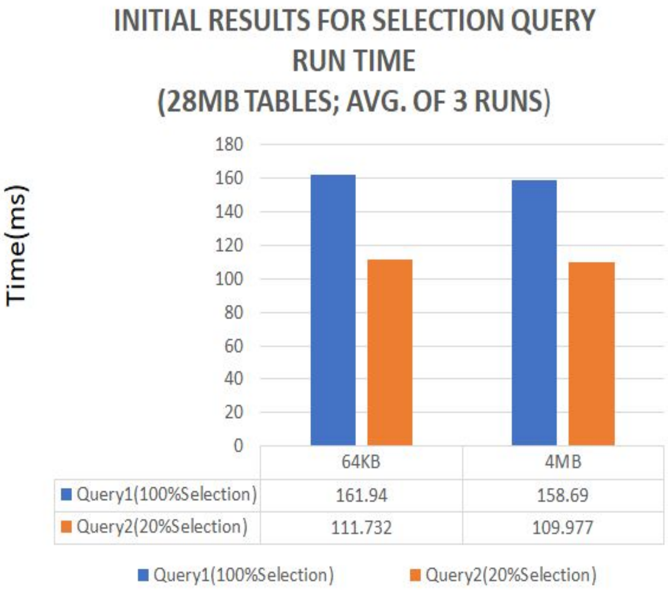
Experiment 2: Benchmark Queries

Query 1: SELECT UNIQUE2, COUNT(*) FROM HUNDREDKTUP GROUP BY UNIQUE2;
Query 2: SELECT TWENTY, COUNT(*) FROM HUNDREDKTUP GROUP BY TWENTY;

Original experiment specifications called for working_memory configurations of 4 MB and 64 KB. We later expanded this design to include 30 MB.

Expected Results:

- Both Query 1 and Query 2 will perform **better when working memory is 4 MB** than when it is 64 KB.
- Query 1 with 100% selection **will be significantly slower** than Query 2 with 20% selection, due to an inefficient hash table created in memory by Query 1.



Experiment 2 Results (contd…)

- Postgres query planner chose disk based group aggregate when the hash table does not fit in memory
- For further investigation, we increased the size of postgres working memory to 30MB to check how postgres “hash aggregate” performs for an indexed attribute

<u>Results:</u>

- When query result size is greater than configured working memory, postgres tends to chose query plan that is optimal rather than generating inefficient hash
- When working memory is equal, and query plan is hash aggregate for both queries (like in case of 30MB) then the execution time for selection of an indexed attribute (100% selection) is almost double the execution time for selection of a non-indexed attribute(20%)
- Comparison of hash aggregation performance on non-indexed attribute for different working memory size and selecting aggregation and group by shows, the size of the working memory does not impact performance. (This could have happened because of sequential scanning in all the cases)

	Work_mem = 64KB		Work_mem = 4MB		Work_mem = 30MB	
	Query1 (GroupAgg.) (Index Only Scan)	Query2 (HashAgg.) (Seq. Scan)	Query1 (GroupAgg.) (Index Only Scan)	Query2 (HashAgg.) (Seq. Scan)	Query1 (HashAgg.) (Seq. Scan)	Query2 (HashAgg.) (Seq Scan)
Average (Excluding min & max execution time)	161.940ms	111.732ms	158.69ms	109.977ms	191.674ms	108.167ms

Experiment 2: Raw Data (Readings in ms)

Query1: SELECT UNIQUE2, COUNT(*) FROM HUNDREDKTUP GROUP BY UNIQUE2;
Query2: SELECT TWENTY, COUNT(*) FROM HUNDREDKTUP GROUP BY TWENTY;

	Work_mem = 64KB		Work_mem = 4MB		Work_mem = 30MB	
	Query1(100% selection)	Query2(20% selection)	Query1(100% selection)	Query2(20% selection)	Query1(100% selection)	Query2(20% selection)
Run1	170.727ms	115.897ms	145.564ms	117.223ms	205.459ms	109.143ms
Run2	145.386ms	106.037ms	173.268ms	105.249ms	187.165ms	110.022ms
Run3	151.672ms	114.508ms	250.324ms	112.961ms	199.507ms	107.775ms
Run4	168.491ms	96.627ms	145.021ms	106.332ms	185.292ms	107.584ms
Run5	165.659ms	114.653ms	157.24ms	110.638ms	188.351ms	103.599ms
Average (Excluding min & max execution time)	161.940ms	111.732ms	158.69ms	109.977ms	191.674ms	108.167ms

Experiment 3: Sizeup

- Explore the query response time of a relation when the relation size differs
- Further analyze the result when the query selects indexed and non-indexed attributes separately
- Expected Results:
  - Use of index scan for indexed attribute selection query; and use of other scan methods (such as seq. scan) for non-indexed attribute selection query
  - When a query has selective predicate as a non-indexed attribute, sizing up data will result in significant difference in execution time between small data size and larger data size

Experiment 3: Benchmark Queries

Query 1: SELECT * FROM HUNDREDKTUP WHERE UNIQUE2 = 99999; (indexed)
Query 2: SELECT * FROM TENMILTUP WHERE UNIQUE2 = 99999; (indexed)
Query 3: SELECT * FROM HUNDREDKTUP WHERE UNIQUE3 = 99999; (non-indexed)
Query 4: SELECT * FROM TENMILTUP WHERE UNIQUE3 = 99999; (non-indexed)

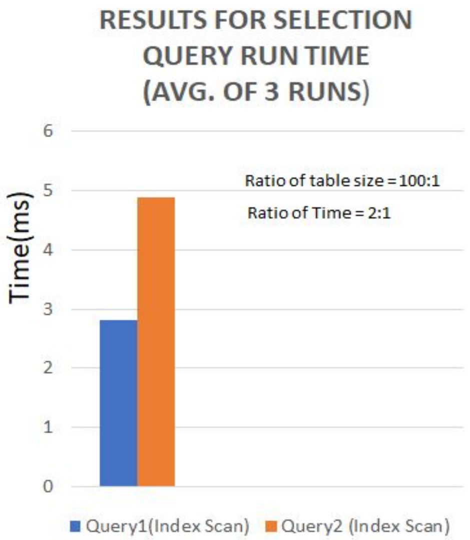
Expected Results:

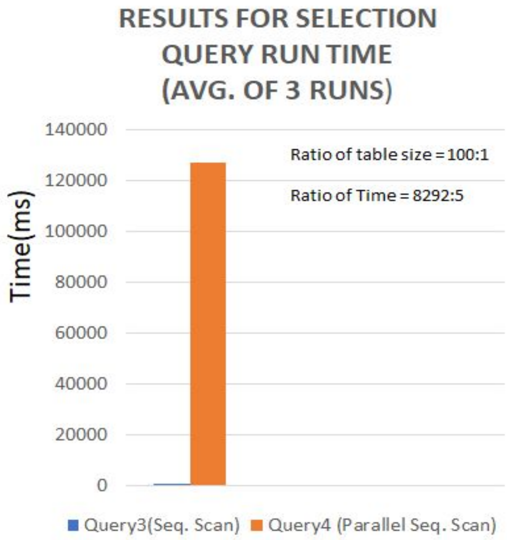
- Between Query 1 and Query 2, execution time for Query 2
  is expected to be longer, but not in the same ratio of the corresponding table sizes.
- Between Query 3 and Query 4, run time will be close to one another(assumption: both queries selection predicate attribute are non-indexed so postgres could possibly perform sequential scan in both queries)

Experiment 3 Results

Results:

- Between Query1 and Query2, execution time of Query1 is less but with comparison to data size increment, run time of Query2 is not as bad
- Between Query3 and Query4 it was expected both the queries will execute sequential scan and the execution time of Query4 will be close to Query3. However, the execution time for Query4 is orders of magnitude larger than Query3.
- The execution time for Query4 is large possibly because it is executed on a very large data set and selection predicate is a non-indexed attribute. So, the query has to run through all the data before giving back the result.





Experiment 3: Raw Data (Readings in ms)

Query 1: SELECT * FROM HUNDREDKTUP WHERE UNIQUE2 = 99999;
Query 2: SELECT * FROM TENMILTUP WHERE UNIQUE2 = 99999;
Query 3: SELECT * FROM HUNDREDKTUP WHERE UNIQUE3 = 99999;
Query 4: SELECT * FROM TENMILTUP WHERE UNIQUE3 = 99999;

	Query1(Indexed)	Query2(Indexed)	Query3(Non-Indexed)	Query4(Non-Indexed)
Run1	1.821ms	12.362ms	78.068ms	127319.687ms
Run2	2.641ms	5.001ms	68.803ms	127293.421ms
Run3	2.866ms	4.480ms	77.083ms	127514.963ms
Run4	2.950ms	4.606ms	78.394ms	126815.828ms
Run5	3.111ms	5.059ms	75.128ms	127282.069ms
Average (Excluding min & max execution time)	2.819ms	4.888ms	76.759ms	127298.392ms

Experiment 4 - Design & Results

Observation: Given an equijoin between a small table that fits in memory and a much larger table, typically **hash joins are preferred** over all other join types.

Central Question: **By manipulating work_mem, can we force the query planner not to choose hash joins?**

Expectation: Merge joins will be preferred when work_mem drops below the size of the smaller table (20.8 MB), or rises above the size of the larger table (208 MB).

Query:

```
EXPLAIN
SELECT * FROM hundredktup T1
JOIN miltup T2 ON T1.unique1 = T2.unique1;
```

Result: We found **no evidence work_mem was considered** when deciding between merge joins vs. hash joins.

Further Information: We tested work_mem values between 64 Kb and 300 MB. Additionally, max_parallel_workers was tested at 1 and 8.

Note: enable_parallel_hash was disabled for the duration of the experiment, as parallel processes provide additional memory.

Additional Result: When the join attribute was altered from `unique1` (no index) to `unique2` (clustered index), merge joins were preferred.

Conclusions

- PostgreSQL runs the clock algorithm to select victims from the shared buffers--this design choice has real, **observable performance consequences**.
- In particular, queries executing immediately after large updates demonstrate **slightly worse performance** than those which do not.
- PostgreSQL periodically writes dirty buffer pages using a checkpoint mechanism; thus, we speculate that queries executing after a large update, but after **allowing a several-second delay**, will be more performant.
- Although our experiments did not demonstrate this, one could presumably avoid the performance consequences of a large update query by choosing a shared_buffers value larger than two times the size of the largest relation.
- PostgreSQL doesn't use hash aggregates if working memory is insufficient; rather, it chooses a different query plan.

Lessons Learned / Further Research

- The PostgreSQL source code has provisions to swap in other eviction strategies than the clock algorithm. It would be an interesting project to try changing the algorithm to--say--LRU.
- PostgreSQL relies on a checkpoint mechanism to make periodic writes to disk; it would be interesting to compile a list of points in the code where a checkpoint may be raised.
- Postgres configuration parameters supports query optimizer choosing ordinary query plans. If that query plan is not optimal, then postgres tend to choose a different plan.
- The query planner does not seem to consider configured memory and maximum worker threads when choosing a query plan--for example, it will choose a plan with 2 worker threads when only 1 thread is available.

Thanks for listening!

Appendix

```
You are now connected to database "wbdb" as user "postgres".
wbdb=# EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
wbdb-# create temp table hundredktuptemp1 as
wbdb-# SELECT UNIQUE2, COUNT(*) FROM HUNDREDKTUP GROUP BY UNIQUE2;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 GroupAggregate (actual rows=100000 loops=1)
   Group Key: unique2
   ->  Index Only Scan using hundredktup_pkey on hundredktup (actual rows=100000 loops=1)
         Heap Fetches: 100000
 Planning Time: 10.988 ms
 Execution Time: 145.564 ms
(6 rows)
```

```
wbdb=# EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
wbdb-# create temp table hundredktuptemp2 as
wbdb-# SELECT TWENTY, COUNT(*) FROM HUNDREDKTUP GROUP BY TWENTY;
                         QUERY PLAN
------------------------------------------------------------
 HashAggregate (actual rows=20 loops=1)
   Group Key: twenty
   ->  Seq Scan on hundredktup (actual rows=100000 loops=1)
 Planning Time: 9.392 ms
 Execution Time: 167.769 ms
(5 rows)
```

```
wbdb=# show work_mem;
 work_mem
----------
 30MB
(1 row)

wbdb=# EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
wbdb-# create temp table hundredktuptemp1 as
wbdb-# SELECT UNIQUE2, COUNT(*) FROM HUNDREDKTUP GROUP BY UNIQUE2;
                          QUERY PLAN
------------------------------------------------------------
 HashAggregate (actual rows=100000 loops=1)
   Group Key: unique2
   ->  Seq Scan on hundredktup (actual rows=100000 loops=1)
 Planning Time: 9.830 ms
 Execution Time: 205.459 ms
(5 rows)
```

```
 work_mem
----------
 30MB
(1 row)

wbdb=# EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
wbdb-# create temp table hundredktuptemp2 as
wbdb-# SELECT TWENTY, COUNT(*) FROM HUNDREDKTUP GROUP BY TWENTY;
                          QUERY PLAN
------------------------------------------------------------
 HashAggregate (actual rows=20 loops=1)
   Group Key: twenty
   ->  Seq Scan on hundredktup (actual rows=100000 loops=1)
 Planning Time: 9.677 ms
 Execution Time: 109.143 ms
(5 rows)
```

```
wbdb=# EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF) SELECT * FROM HUNDREDKTUP WHERE UNIQUE2 = 99999;
                                QUERY PLAN
--------------------------------------------------------------------------------
 Index Scan using hundredktup_pkey on hundredktup (actual rows=1 loops=1)
   Index Cond: (unique2 = 99999)
 Planning Time: 4.185 ms
 Execution Time: 1.812 ms
(4 rows)
```

```
wbdb=# EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF) SELECT * FROM TENMILTUP WHERE UNIQUE2 = 99999;
                                QUERY PLAN
----------------------------------------------------------------------------
 Index Scan using tenmiltup_pkey on tenmiltup (actual rows=1 loops=1)
   Index Cond: (unique2 = 99999)
 Planning Time: 17.535 ms
 Execution Time: 12.362 ms
(4 rows)
```

```
wbdb=# EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF) SELECT * FROM HUNDREDKTUP WHERE UNIQUE3 = 99999;
                     QUERY PLAN
------------------------------------------------
 Seq Scan on hundredktup (actual rows=1 loops=1)
   Filter: (unique3 = 99999)
   Rows Removed by Filter: 99999
 Planning Time: 11.917 ms
 Execution Time: 78.068 ms
(5 rows)
```

```
wbdb=# EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF) SELECT * FROM TENMILTUP WHERE UNIQUE3 = 99999;
                          QUERY PLAN
------------------------------------------------------------
 Gather (actual rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on tenmiltup (actual rows=0 loops=3)
         Filter: (unique3 = 99999)
         Rows Removed by Filter: 3333333
 Planning Time: 12.157 ms
 Execution Time: 127319.687 ms
(8 rows)
```