Sandy Wood, Sara Alotaibi

CS 487

23 February 2021

Database Benchmarking Project – Part 2

System Research

We have researched the following parameters in Postgres' resource consumption and query planning configurations. We feel that they will be useful in manipulating the system's use of different memory parameters to examine how these affect Postgres' performance.

seq_page_cost (floating point)

This parameter changes the cost that the system applies to fetching a page that is part of a sequential fetch from the disk. Such instances where this would be used is when the system is pulling a large amount of sequential data from the disk, like when SELECT * is used. The default for this value is 1. The costs in the postgres system are evaluated based on their relation to one another rather than an absolute scale.

random_page_cost (floating point)

This parameter changes the cost that the system applies to disk fetches that are not sequential. This can apply to index scan. The default value for this attribute is 4, despite the cost of fetching a random page from disk being much greater than 4 times the cost of a sequential fetch. This is because of the 90% assumption – that 90% of disk pages being requested will be in cache. The difference between these two values determines how much the system will perform index scans; a higher random page cost will cause the database to choose an index scan less, while a value closer to seq_page_cost will cause the system to choose index scans more.

shared_buffers (integer)

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, so this parameter does not have much added performance at settings above 40% of the machine's main memory. The database will also make use of the operating system's disk caching.

work_mem (integer)

work_mem sets the amount of memory that postgres dedicates to tasks such as sorting (order by, distinct) and hash tables. This amount of memory will be used for each operation, so a value of 4MB results in 4MB being used by each hash table, sorting operation, etc. The default is 4MB.

max_worker_processes (integer)

Sets the maximum number of background processes that the system will use. This parameter controls how many process are available, the max_parallel_processes parameter controls how many of these processes can run in parallel, while max_parallel_maintenance_workers and max_parallel_workers_per_gather control how many processes the system will allow to run in parallel for maintenance tasks and parallel queries, respectively. Queries that benefit from parallelism are less common than queries that can gain no benefit from being run in parallel.

Performance Experiment Design

Experiment 1: shared_buffers modification

i. What performance issue are you testing?

This test examines the performance of PostgresQL when the shared_buffers parameter is set to different percentages of the total available system memory.

ii. What data sets will you include in the test?

We will be using a 1,000,000 tuple relationship for this test

iii. What queries will you run? (provide the SQL)

The query that will work best for this experiment is one that needs to load many pages from disk onto memory, but will not write-back to disk, as a random seek back to disk will muddy results.

10% Selection:
SELECT * FROM ONEMTUP
WHERE unique1 BETWEEN 100,000 AND 200,000

1% Selection:
SELECT * FROM ONEMTUP
WHERE unique1 BETWEEN 100,000 AND 110,000

iv. What parameters will you use for this test? How will the parameters be set/varied?

We will be modifying the shared_buffers parameter. This parameter will be changed respective to the memory that is available on the VM. We will run tests at 10%, 20%, 40%, 60%, and 80% of memory.

v. What results do you expect to get?

We expect to see the time for the system to select a query increase as memory allocated to the buffers increase. However, we expect that 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.

Experiment 2: Join algorithm performance

i. What performance issue are you testing?

In this performance experiment we are going to test the performance of different join algorithms.

ii. What data sets will you include in the test?

For this experiment we will use Two tables tenKtup1 and tenKtup2 in PostgreSQL database and each one of these tables have 1,000,000 tuples

iii. What queries will you run? (provide the SQL)

Query will retrieve rows from table tenKtup1 for which unique1 values exist in table tenKtup2.

SELECT * FROM tenKtup1
WHERE unique1 in (SELECT unique1 from tenKtup2)

This query will do what the previous query did in addition to Enable_nestloop to off.

```
Set Enable_nestloop = off
SELECT * FROM tenKtup1
WHERE unique1 in (SELECT unique1 from tenKtup2)
```

This query will do what the first query did in addition to Enable_mergejoin to off.

```
Set Enable_mergejoin = off
SELECT * FROM tenKtup1
WHERE unique1 in (SELECT unique1 from tenKtup2)
```

iv. What parameters will you use for this test? How will the parameters be set/varied?

In the second query we turned off Enable_nestloop parameter because when we turn off this parameter it will have other options of hash and merge join only.
In the third query we turned off Enable_mergejoin parameter because it will force us to use hash join in the query plan.

v. What results do you expect to get?

Result for first query: it will give a nested loop join in query plan. Because both tables have a large number of rows and it has more than 10% selectivity. Both tables tenKtup1 and tenKtup2 will be scanned sequentially. So it will give a nested loop join in the query plan.

Result for second query: when we turned off Enable_nestloop parameter then it will use merge join in query plan. Here the two tables are very large so it will not fit in memory. So it will use merge join in the query plan, and it will improve performance of the query.

Result for third query: when we turned off Enable_mergejoin parameter then it will force to use hash join in query plan. Here both tables are very large so it will divide data into buckets and will perform hash join. Moreover, it will make performance worse than merge join as tables are very large and not fitting in memory.

Experiment 3: Change work_mem on joins

i. What performance issue are you testing?

We will be testing the performance of postgres hash joins when given very small to very large work_mem values.

ii. What data sets will you include in the test?

The OneMTup and 100KTup set will both be used. The hash table will be built on the smaller of the datasets. The join attribute will be onePercent, so that the hash table created will have a maximum of 100 entries.

iii. What queries will you run? (provide the SQL)

```
SELECT M.unique1, K.unique1
FROM OneMTup M, OneKTup K
WHERE K.onePercent = M.onePercent
```

iv. What parameters will you use for this test? How will the parameters be set/varied?

We will be adjusting the work_mem parameter from a small value (1MB) to larger values (1GB) at significant steps to see how psql's hash join performs.

v. What results do you expect to get?

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. However, we are also thinking that if the required memory is never provided, we will not see an optimal performance of the hash join

Experiment 4: Change work_mem on aggregate queries

i. What performance issue are you testing?

We will be testing the performance of postgres aggregate queries when allowed to have very small and very large work_mem values.

ii. What data sets will you include in the test?

We will be using the ONEMTUP relation for this test. It is a sufficiently large relation for our purposes. We will be using the .onePercent attribute in order to group the data into 100 buckets of equivalent size.

iii. What queries will you run? (provide the SQL)

```
INSERT INTO TMP
SELECT SUM (ONEMTUP1.unique3) FROM ONEMTUP1
GROUP BY ONEMTUP1.onePercent
```

iv. What parameters will you use for this test? How will the parameters be set/varied?

We will be adjusting the work_mem parameter from a small value (1MB) to larger values (1GB) at significant steps to see how psql's aggregate queries perform with access to different amounts of memory.

v. What results do you expect to get?

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 used to perform the aggregate query appears simultaneously in memory. 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.

Lessons Learned

In the second part of the project we have a chance to learn about Postgres parameters and optimization options. Also, we notice the big rule that selectivity plays in the query plan selection and the effects of data size on query optimizers. We also learned how we can get better query performance by tweaking configuration parameters like work_mem. Before this project our knowledge was limited in query execution without thinking of what query optimizer actually does but now we know how configuration parameters can affect the behavior of database systems. Cost parameters can greatly affect the way that postgres will choose query plans and estimate the costs of those plans. Additionally, there is no one correct configuration of parameters. Rather, each situation has different types of queries and different stresses - an analysis of these stresses and the most common types of queries requested can greatly impact which values these parameters have. Considerations such as operating system, number of concurrent users, server hardware, storage hardware, and usage patterns can all have

large implications for configuring optimum parameters. We also did some research into parallel queries, and learned when we need to consider their usage, and when they will never run. We decided not to experiment with any of these parameters, but we still found it valuable to determine when postgres will take advantage of parallelism to run a query, and what the consequences of parallelism are on memory.