CS 487/587 Database Implementation
Winter 2021
Database Benchmarking Project - Part 2

By: Pooja & Utsav

1. About the System:

We used the PostgreSQL database system for this project. We used this system due to a number of reasons.

1) The familiarity with the system. We've used postgreSQL for previous courses and find it comfortable to play around with different queries and analyze their performance.

2) It is easier to use the same system to compare the performance than using two different systems. We can focus on various parameters in this system such as cache, memory, enabling joins, etc while the basic parameters such as the Operating system and other hardware requirements and resources remain constant.

3) There are a lot of resources and their usage is growing. It is robust with high performance and multitasking capabilities. Since it's an open-source database system there are active threads of communities and different users who develop and post their results and also answer some other queries.

4) There are certain ways to turn off and on the parameters that a query optimizer will use and we can observe the change in time and cost of doing so and also be intrigued by the query optimizer's choice of plan.

2. System Research

enable_hashjoin:

This enables or disables the query planner's use of hash join and by default, it's on. It's a frequently used join algorithm where a hash table is first built using inner relation records and the hash key is calculated based on the join attribute and then the probing phase where the outer record relation is hashed based on the join attribute to find a matching entry in the hash table. It is mostly used on relations that have not been sorted on the join attribute.
The cost is generally low compared to other joins and also based on the query and if the relation fits in memory. The cost is significantly higher when accessing the disk. It is the join algorithm that can only handle equi joins.

enable_mergejoin:

______This enables or disables the query planner's use of merge join and by default, it's on. This algorithm is only used if both the relations are sorted and the join attribute operator '=' is used. If a table can be retrieved in sorted order based on the index when the relation is already sorted on the join attribute, then the merge join method is used and it should perform better than hash join.

work_mem:

______The amount of memory that is used by internal sort operations and hash tables before writing it to the temporary disk files and the default valis is 4 MB as specified in the work_mem as per the Postgres resource consumption documentation. A complex query may use a number of sorts and hash operations running in parallel each of which uses memory, and hence we have to keep that in mind while setting the value of work_mem. Hence, setting up large work memory can result in worse performance when many queries run concurrently since each of them will try using up huge amounts of memory which cannot happen when there are too many processes.

Sort operations happen when an ORDER BY, DISTINCT, and a merge JOIN is used. Hash tables are used in the case of hash joins, hash-based aggregations and hash-based processing in sub queries.

Incrementing or decrementing their values results in varying performance for sort/hash queries.

effective_cache_size :

Should be set to an estimate of how much memory is available for disk caching by the operating system and within the database itself, after taking into account what's used by the OS itself and other applications. This is a guideline for how much memory we expect to be available in the OS and PostgreSQL buffer caches, not an allocation! This value is used only by the PostgreSQL query planner to figure out whether plans it's considering would be expected to fit in RAM or not. If it's set too low, indexes may not be used for executing queries the way we'd expect. The setting for shared_buffers is not taken into account here--only the effective_cache_size value is, so it should include memory dedicated to the database too.

Setting effective_cache_size to 1/2 of total memory would be a normal conservative setting, and 3/4 of memory is a more aggressive but still reasonable amount. You might find a better estimate by looking at your operating system's statistics. On UNIX-like systems, add the free+cached numbers from free or top to get an estimate. On Windows see the "System Cache" size in the Windows Task Manager's Performance tab. Changing this setting does not require restarting the database (HUP is enough).

Random_page_cost:

Sets the planner's estimate of the cost of a non-sequentially-fetched disk page. The default is 4.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name.Reducing this value relative to seq_page_cost will cause the system to prefer index scans; raising it will make index scans look relatively more expensive. We can raise or lower both values together to change the importance of disk I/O costs relative to CPU costs.

3. Performance Experiments

Performance Experiment 1:

Performance Issue:

We analyze the impact of various join algorithms on the query execution time.

Data set:

The dataset we use here is a join on onektup and tenktup which consists of thousand and ten thousand records respectively.

Query:

In this query, we extract all the records of onektup and tenktup on which a join operation is performed. All the records where tenktup's unique2 values are lesser than 2000 are fetched and analyzed.

Merge join takes less time:

EXPLAIN ANALYZE SELECT * FROM onektup,tenktup1 WHERE onektup.unique2 = tenktup1.unique2 AND tenktup1.unique2 < 2000;

Hash join takes less time:

EXPLAIN ANALYZE SELECT * FROM tenktup1,tenktup2 WHERE tenktup2.unique1 = tenktup1.unique1 AND tenktup1.unique1 BETWEEN 7000 AND 10000;

Parameters:

The parameters we'll be changing and experimenting on would be enable_mergejoin and enable_hashjoin.

Results Expected:

When we execute this query it uses Merge join by default and when I switch the parameter of merge join off we can see the results of hash join and when I switch it off i.e when both merge and hash join is off it switches to another join here which is probably a nested loop join. For the above query, merge join performs better because we already have the first 2000 records already sorted and we also used clustered index. This implies that the cost of the sort can be lowered when the data sets can be accessed in sorted order via an index. When varying the querying attribute and using a number of comparison parameters we can observe hash join performs better when the hash table can be held in memory.

Performance Experiment 2:

Performance Issue:

There could be a difference in the performance when the work_mem is increased or decreased based on the no of connections and also the sort operation performed on the relations.

Dataset:

We take relationships of all sizes i.e 1k, 10k, 100k tuples to study the difference in performance.

Query:

EXPLAIN ANALYZE SELECT unique2,stringu1 FROM hundredktup ORDER BY stringu1;

EXPLAIN ANALYZE SELECT tenktup1.unique2,hundredktup.string4 FROM tenktup1, hundredktup WHERE tenktup1.string4 = hundredktup.string4 ORDER BY string4;

Parameters:

We'll change work_mem and set it to vary between 1MB and 128 MB.

Expected Results:

We are expected to see improved performance with an increase in work memory where sorting and hashing happens, especially when dealing with relations that don't fit in memory.

Performance Experiment 3:

Performance Issue

This test explores the relationship between the effective_cache_size and how the setting of this parameter affects the performance of queries.

effective_cache_size should be set to an estimate of how much memory is available for disk caching by the operating system and within the database itself, after taking into account what's used by the OS itself and other applications. This is a guideline for how much memory we expect to be available in the OS and PostgreSQL buffer caches, not an allocation! This value is used only by the PostgreSQL query planner to figure out whether a plan would fit in RAM or not. If it's set too low, indexes may not be used for executing queries the way we'd expect. The setting for shared_buffers is not taken into account here--only the effective_cache_size value is, so it should include memory dedicated to the database too.

Setting effective_cache_size to 1/2 of total memory would be a normal conservative setting, and 3/4 of memory is a more aggressive but still reasonable amount. You might find a better estimate by looking at your operating system's statistics. On UNIX-like systems, add the free+cached numbers from free or top to get an estimate. On Windows see the "System Cache" size in the Windows Task Manager's Performance tab. Changing this setting does not require restarting the database (HUP is enough).

Data Set

Three different relations of size 10k,100k and 1000k tuples are used in this performance test.

Query

The query takes tuples that are indexed but tries to see how the effect of this parameter is taken into account by the query optimizer. Basically we want to see that the query optimizer doing sequential scans when the effective cache size is set to too low and the query optimizer resorting to a much faster index scan when more disk cache space is available.

set effective_cache_size= varied(1MB to 128 MB)

Parameters:

The parameter that we would be toggling here is effective_cache_size

Results expected

When performing a scan the disk cache will be taken into account by the query optimizer to decide if it can do an index scan or not, we expect to see less time taken by the query optimizer when there is more than required disk cache available.

Performance Experiment 4:

Performance Issue

This test explores the relationship between random_page_cost and how the setting of this parameter affects the performance of queries.

Reducing this value relative to seq_page_cost will cause the system to prefer index scans; raising it will make index scans look relatively more expensive. You can raise or lower both values together to change the importance of disk I/O costs relative to CPU costs, which are described by the following parameters.

Random access to mechanical disk storage is normally much more expensive than four times sequential access. However, a lower default is used (4.0) because the majority of random accesses to disk, such as indexed reads, are assumed to be in cache. The default value can be thought of as modeling random access as 40 times slower than sequential, while expecting 90% of random reads to be cached.

If you believe a 90% cache rate is an incorrect assumption for your workload, you can increase random_page_cost to better reflect the true cost of random storage reads. Correspondingly, if your data is likely to be completely in cache, such as when the database is smaller than the total server memory, decreasing random_page_cost can be appropriate. Storage that has a low random read cost relative to sequential, e.g. solid-state drives, might also be better modeled with a lower value for random_page_cost.

Data Set

Three different relations of size 10k,100k and 1000k tuples are used in this performance test.

Query

The query will be performed on tuples which already have an index on them and will try to see if increasing the value of random_page_cost affects the performance and will try to see how the query optimizer changes its plans accordingly and will try to see where this might be effective.

set random_page_cost=100

Parameter

The parameter we would be toggling here is random_page_cost

Results expected

When we lower the value of random_page_cost it means that all the pages can be cached in the memory which means the query optimizer is more likely to use an index if it can instead of doing a heap scan.

Performance Experiment 5:

Performance Issue

This test explores how the query optimizer benefits from a clustered index vs a non-clustered index.

Clustered index is a sort of index which is stored in memory but also the tuples in the table are sorted on the basis of the clustered index. This provides an unique benefit to the query optimizer and the query optimizer can go to a particular index in the table if the query is using a clustered index vs a non-clustered index.

Data Set

Three different relations of size 10k,100k and 1000k tuples are used in this performance test.

Query

The query will be performed on tuples which already have a clustered and a non-clustered index on them and will try to see how the query optimizer changes its plans accordingly when using a clustered index vs a non clustered index.

Parameter

The parameter we would be using here is unique2 which is a clustered index and we will be using unique1 which is a non-clustered index.

Results expected

We expect to see a higher execution time when we query a non clustered index vs a clustered index.

Performance Experiment 6:

Performance Issue:

In an aggregate function the tuples are partitioned into subsets by the group by construct. Now on each of the partitioned subsets an aggregate operation is performed. These aggregate operations include MIN scalar aggregate queries, MIN aggregate function queries and SUM aggregate function queries each of which can be executed two ways. To use any non clustered index could result in scanning of the entire relation and each data page would be accessed several times.

Data Set

A 10k, 100k tuple size relation could be used.

Query

10k tup relations can be used for testing and a MIN operation can be performed to get a scalar value on the clustered index unique2 and also a MIN on unique3 can be performed when it is partitioned and grouped by the attribute onepercent (which is 0.01 * 10,000 = 100 partitions). And also a SUM of unique3 could be performed instead of just MIN.

```
SELECT MIN(tenktup1.unique2) FROM tenktup1;

SELECT MIN(tenktup1.unique3) FROM tenktup1 GROUP BY tenktup1.onePercent;

SELECT SUM(tenktup1.unique3) FROM tenktup1 GROUP BY tenktup1.onePercent;
```

Parameter

We could vary the clustered index and non clustered index and check for the delay in execution time and also the selectivity factor which can be onePercent, tenPercent, etc. We could also sort on the partitioning attribute and then collect the result.Another approach would be hashing on the partitioning attribute.

Results expected

If any secondary, non-clustered index is used on the partitioning attribute, it would slow down the querying as a complete scan of the relation has to be performed. If we ignore the index, sort on the partitioning attribute then the scan will be faster which results in improved performance.

Performance Experiment :

Performance Issue & Results expected:

Sizeup Experiments:

Query response time should grow linearly with size of the input and result relations. Different selectivity factors for example 1%, 10% of the relation size can result in different results and in execution time based on the size of the relation chosen which could be 10 thousand,1million or even 10 million.The 10% selection using a clustered index is another example where increasing the size of the input relation by a factor of ten results in more than a ten-fold increase in the response time for the query. We need to consider the impact of seek time on the execution time of the query. The cost of writing each output page is higher since the disk heads must be moved from current position over the input relation to a free block on the disk. But, this does not happen in 1% non-indexed selection since the resultant relation fits in the buffer pool and it can be written to the disk sequentially at the end of the query. Execution of queries through a clustered index improves the performance.

Data Set

A 10k,100k, 1million tuple relation could be used for this performance test.

Query

SELECT * FROM tenktup1 WHERE unique1 BETWEEN 700 AND 1700

4. Lesson Learned:

1) We have learned how just changing a single parameter can have an impact on performance. For example, a clustered index may take less time to fetch queries than an unclustered index but, again the reverse is possible too. Forcing the query optimizer to pick a plan than what it originally did sometimes enhanced the performance and it was interesting to analyze why the optimizer would have chosen a particular plan.

2) We understood the real world impacts of a clustered index vs a non-clustered index and why a good index always speeds up your execution time.

3) We understood multiple parameters that postgresql has and understood the practical benefits of tuning these parameters to get better results.

4) We would have loved to perform speedup experiments but we didn’t have that many processors at our disposal to perform speedup experiments.

5) With time what we have observed is that the performance of the single user systems has not changed drastically by enabling or disabling any parameter but even those slight differences would become big ones in a multiuser environment. And hence we understood why gaining the most optimal performance on a single user system is more beneficial as it would probably be an indicator to how the system would perform in a multi-user environment.