Wisconsin Benchmark Pt 2: Benchmarking Postgres

Logan Ballard

05/15/2019

1. **Background**

   Due to its ubiquity and my own familiarity with the system, I have chosen to benchmark postgres against itself. Postgres is a very flexible and well-supported database system which has a lot of community and enterprise backing behind it. It will behoove me to learn much mroe of its ins and outs if I wish to use it professionally, which seems very likely given how widespread its adoption is.

   In addition to business reasons for choosing postgres, it is useful because it has a large amount of configurability. This ability allows for near-infinite tweaking of performance to maximize different attributes of the database. It also can be run locally without much setup, which allows for rapid iteration on various ideas. Through tools like *pgAdmin* and integration with familiar programming languages like *python*3 with the *psycopg*2 library, I was able to interface easily with the database and tune performance to my liking. This allows for more in-depth and meaningful exploration of the actual data and concepts without extended ramp-up to the system.

2. **The System**

   I examined a few parameters that are involved in tuning postgres performance. They came in two flavors: Memory Parameters and Query Planner Parameters.

   (a) Memory Parameters

      i. *work_mem* - change performance of hash joins and sorts

   (b) Query Planning Parameters

      i. *enable_hashjoin* - Will want to find queries where a hash join is preferable and attempt to show performance of them improving or degrading.

      ii. *enable_mergejoin* - Will want to find queries where a merge join is preferable and attempt to show performance of them improving or degrading. Maybe create a hierarchy of join types?

      iii. *enable_sort* - All sorting steps will be discouraged. I would expect this to degrade performance significantly in ORDER BY and GROUP BY as well as sort-merge join queries.

      iv. *geqo_effort* - This parameter sets the amount of time that postgres will spend looking for the ideal query plan. It will be most effective in complicated queries (I suspect)

3. **The Experiments**

   For these experiments, I will start with attempting a simplistic version of the test. From there, I will move on to something more complicated and then finally perform the most intricate version of the test. With each experiment I will measure the time taken for the query, join, insert, or update to execute.

   (a) **Experiment 1: The 10% Rule of Thumb**

i. This test explores when it is good to use an unclustered index vs. not using an index vs. using a clustered index
ii. Use a 100,000 tuple relation (scaled up version of TENKTUP1)
iii. Use Wisconsin Bench queries 2, 4 and 6. Run queries 2, 4, and 6 on the same dataset. Query 2 should be run without an index on unique2, Query 4 should be run with a clustered index on unique2, Query 6 requires an unclustered index on unique1
iv. No parameters changed in this test
v. I would expect that the rule of thumb is roughly correct. I would think that once you reach the 10% selectivity factor, there will be little gain from using an index. However, I believe that RDBMSs have made significant strides in the past years, and I can only assume that they have created baked-in optimizations that are transparent to the end user. Therefore, I suspect that the selectivity rules are creeping steadily upward.

(b) **Experiment 2: Join Algorithms - Hash vs Sort/Merge**

i. This test aims to measure the performance of different types of joins across different relations using different algorithms. It is posited that hash joins are superior with a small relation is joined to a larger relation, and a sort/merge join performs better on two comparatively large relations.
ii. For this experiment I will use purely data from the Wisconsin Benchmark. I will use the ONEKTUP, TENKTUP, BPRIME, and the newly created HUNDREDKTUP (from experiment 1) for testing purposes.
iii. I will use queries 10 and 13 initially. This is a large table joining a smaller table, once with an index and once without. I will use Query 15 with the prescribed TENKTUP tables as well as the HUNDREDKTUP created for experiment 1. Finally, I will check out Query 17 in order to test hash join.
iv. I will be changing the boolean flags: *enable_hashjoin* and *enable_mergejoin*.
v. I expect that I will see pretty drastic performance changes between enabling and disabling the different parameters. The more interesting changes will be in situations where hash join were ordinarily be faster than a sort/merge join. These performance differences will be tested with query 17, because ONEKTUP should be able to fit completely in memory. So when hash join is enabled, I would expect performance to improve over sort/merge join. For other relations, especially as the relations become very large (10K, 100K, possibly even something like 1M if my machine allows it), I would expect sort/merge join to outperform hash join.

(c) **Experiment 3: Aggregation**

i. This test aims to measure the performance of aggregation with relevant parameters tuned to different values. The spirit of this experiment will be testing at which point an aggregation step becomes unusably inefficient. For two of the relations, ONEKTUP and TENKTUP, I'll be using a GROUP BY aggregation operator on different columns. Because these tables have various levels of uniqueness throughout their columns, it is easy to determine which columns should yield better performance than others. The three columns I will group by are, in descending order of predicted performance, *ten*, *evenOnePercent*, and *unique*3.
ii. For this experiment I will use purely data from the Wisconsin Benchmark. I will use the ONEKTUP and the TENKTUP relations. If there is time and the results look to be interesting, I will also use the HUNDREDKTUP relation.
iii. Queries will be similar to Query 22 and 25 from the Wisconsin Benchmark, but will not use a summation function. Instead, I will swap out this function for a COUNT(*) function to try and isolate the effects of the aggregation (don't add additional work of finding the minimum value of a group).

iv. There are two parameters I will be tweaking to try to get interesting results from this test: work_mem and enable_sort. Enable_sort will be turned off and aggregations performed. Once it is re-enabled, I will test a sliding scale of values for work_mem to see if the amount of memory allocated scales linearly to the performance of the aggregation.

v. I expect the disabling of enable_sort to have a drastic effect on performance. Because the aggregation operations will definitely want to be using sorting functions, if their query plans are steered away from a sorting operation, it will result in them likely using hashing or some yet unknown method to aggregate. I will expect performance to significantly degrade with this parameter turned off. I expect work_mem to have a non-linear effect on aggregation. I expect that there will be certain "checkpoints" where working memory hits a value that allows a relation of a certain size to fit completely in memory and be operated upon without IO slowdown. These "checkpoints" will have drastic improvement over the previous marker but I suspect that performance will not vary in a meaningful way in between these points.

(d) **Experiment 4: Query Optimization**

i. For this experiment, I'd like to figure out just how much the query optimization time that postgres gives itself actually helps with performance. I'd like to run a bunch of rounds of a few queries, each with the tuning set to different levels just to see if the improvement scales roughly linearly with the amount of time that the optimizer spends trying to optimize it.

ii. Because the optimizer is likely only used when there can be a variety of different joins, I'd like to create a three-way and four-way join between the 1k, 1k, 10k, and (another copy ofthe) 10k different tables.

iii. Join queries, based on the ones from the Wisconsin benchmark.

iv. *geqo_effort* will be tuned accordingly. There are several different integers corresponding to the amount of time that postgres will take trying to find the best possible query tree for a certain join.

v. I expect that for simple queries that involve just a single two-way join, this number will not matter a whole lot. However, as the amount of tables involved and the complexity of the joins themselves goes up, I suspect that this number will begin to be more and more important.

4. **Lessons Learned**

(a) As always, I learned the lesson that I would hope that I learned each and every time that I do one of these projects: start earlier. It is always more difficult to play catch up than it is to fine tune as you get further out in the timeframe. Someone much smarter than me once told me of the 90/90 rule, where "the first 90% of the project takes 90% of the time, the last 10% takes the other 90% of the time." This was the case in this project as well.

(b) Another less learned was to pick an option early and stick with it. I spent a lot of time waffling between the two choices that I *could've* made (comparing systems vs comparing parameters), and put a lot of initial research into both without firmly committing to either one. When I first approached this assignment, I figured that by gathering all the data beforehand and then carefully weighing my options, I could make the absolute best decision. What actually happened was that I ended up taking a large amount of time to consider two choices whose differences were ultimately not gigantic. It would've been a much more effective use of time to simply pick one and stick to it. Eventually I decided on testing the postgres parameters over Postgres Vs MySql because of a pretty inconsequential factor: the fact that I could work on postgres locally without an internet connection, and mySql would've required that I had access to the school's linuxlab machines.