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

      ii. *temp_buffers* - ????

   (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. asdfasdfasdfasdfdsafdsa

(b) **Experiment 2: Joins**

i. This test aims to measure the performance of different types of joins across different relations using different algorithms.
ii. Use the 1ktup joined with 1ktup, 10ktup joined with 10ktup
iii. Run several joins - inner join on some predecate, left join, and joins not based on equality (join where id ¿ 10)
iv. Change hashjoin and mergejoin allowed
v. asdfasdfasdfasdfdsafdsa