Rationale for Choosing PostgreSQL

Most of my experience with databases has been in Oracle (v. 6 - 11g). My experience with other database management systems has been limited to using MySQL for instruction in introductory database courses and PostgreSQL in CS 586 at PSU. I am interested in learning more about PostgreSQL, both for the sake of learning and for possible use in future courses I teach.

Also, when doing some research to identify database management systems to use for this project, I came across many online articles about Oracle vs. PostgreSQL. If so many people want to make comparisons between the two, I figure I should know more about PostgreSQL before I make any professions about the supremacy of one or the other.

System Research about PostgeSQL

All work for this project was done on Windows 10 with PostgreSQL 11 installed on a machine with the following specifications:

> Processor: Intel(R) Core(TM) i5-7200U @ 2.50GHz 2.70 GHz

> Memory: 8 GB

> System: 64-bit OS, x64-based processor

Join Algorithms

PostgreSQL has three join algorithms: nested loop, merge, and hash.

Indices

PostgreSQL has support for the following indices:

1) GIST (Generalized Search Tree) index - something of a framework for other indices, which can be adapted for use with many data types and in combination with other indicies.

2) SP-GIST (Space-partitioned Generalized Search Tree) index - used to partition space while searching spatial data.

3) Hash index - standard hash algorithm.

4) B-Tree index - standard b-tree algorithm.

5) BRIN (Block Range Index) index - used to store information about blocks of data.

6) GIN (Generalized Inverted Index) index - used with arrays of data or multipartite data types, such as structures.

The default index algorithm is the B-Tree. PostgreSQL does not support clustered indices.

Buffer Pool Structure & Size

The shared buffer pool is adjustable in size, with a default size of 128 MB. Its size is limited by the amount of available RAM and can be set to up to 40% of the size of memory.

Buffer pool slots are 8 KB in size (page size).

Query Performance

During execution of a query in PostgreSQL, a number of statistics are gathered, which can be used to measure performance. To limit overhead during query execution, configuration of the postgresql.conf file allow for inclusion or exclusion of specific performance data collection. All of the statistic-gathering is accomplished by the PostgreSQL Statistics Collector, an internal subsystem that can track data about both disk block and individual rows.

Areas of Interest for Benchmarking

For purposes of this project, I chose to explore two areas: joins and indices (in inserts and in queries with aggregate functions). The queries are in the file, benchmarkQueries.sql.

Joins

Test 1: I wrote three queries to test for joins: one that used an inner join in the FROM statement, one that used an inner join as a subquery in the SELECT statement, and one that used two inner joins in the FROM statement. I was particularly interested to see how planning and execution times would vary for the three queries. Also, I was curious to see which join algorithms would be used.

Indices

Test 2: I ran one query that copied the values in tenktup1 to another table, the first time without any indices being created and the second time after creating an index on the unique2 attribute. What I hope to see is how the overhead of planning time when using an index impacts the overall speed of execution.

Test 3: Similar to Test 2, I wrote a single query to test once without an index (in this case, without an index on the attribute being aggregated) and once with. Again, I wanted to explore how use of an index affects planning and execution time.

Results

Test 1:

Description: INNER JOIN on 2 tables, INNER JOIN in FROM
Query Plan: Hash joins and sequential scans, with a total of 4 loops
Planning Time: 1.844 ms
Execution Time: 15.278 ms

Description: INNER JOIN on 2 tables, INNER JOIN in SELECT (subquery)
Query Plan: sequential & index scans, aggregation, 10000 loops
Planning Time: 0.214 ms
Execution Time: 31.205 ms

Description: INNER JOIN on 3 tables, INNER JOIN in FROM
Query Plan: nested loop, merge join, index scan, sort (quicksort), index only scan
Planning Time: 1.861 ms
Execution Time: 2.515 ms

Test 2:

Description: INSERT without index on attribute used in WHERE clause
Query Plan: index scan
Planning Time: 0.279 ms
Execution Time: 0.128 ms

Description: INSERT with index on attribute used in WHERE clause

Query Plan: index scan

Planning Time: 3.105 ms

Execution Time: 0.128 ms

Description: aggregate without index on attribute in GROUP BY clause

Query Plan: hash aggregate, sequential scan

Planning Time: 0.096 ms

Execution Time: 3.815 ms

Description: aggregate with index on attribute in GROUP BY clause

Query Plan: hash aggregate, sequential scan

Planning Time: 1.485 ms

Execution Time: 4.731 ms

Analysis & Conclusions

In general, I found that the inclusion of additional indices does directly lengthen the planning time of a query. Also, accomplishing an inner join using a subquery takes a significant amount of time more than for a direct inner join. Neither of these conclusions is surprising. However, the sampling size used for the benchmark testing was quite small, so drawing conclusions based on these tests is not advised. Rather, the results lead me to want to perform other tests on larger data sets, more demanding queries, while controlling configuration parameters to hone in on specific consequences for query actions, and doing similar queries using different attributes.