Tyler Race

CS487 - Project pt. 2

For part 2 of the project, I decided to go with option 2 (using the same system but tweaking system parameters to show the differences in performance). I chose this option because I have a locally installed copy of Postgres on a virtual machine, and thus tweaking system parameters is extremely easy and safe. It will also be interesting to see how much performance can be increased or if it can be, when changing systems values away from the defaults. Also the computer where postgres is installed, has a solid state disk, which could impact the results significantly. The benchmark paper often make remarks in regards to disk seek times.

**Some Postgres configuration options:**

shared\_buffer: This setting configures the amount of memory the database server uses for shared memory buffers. The shared memory buffers are predefined areas of memory that the database uses for storing pages of data (aka buffer pool). The value default is 128MB, and is best set in the 64MB - 512MB range on Windows systems, but should reflect roughly 25% of available system memory.

enable\_hashjoin: This setting allows the query planner to use the hashjoin algorithm. A hash join creates a temporary hashtable one one side, then hashes values of the other side to find matches. The hashing mechanism itself is what makes this join perform well. If the join predicate contains many similar values however, hashing will experience many collisions, lowering performance.

enable\_sort: This setting enables or disables the query planner’s ability to perform explicit sorting steps. These could include sorting the index (like unique1) to perform more optimally. The default for this setting is turned on.

**Performance tests:**

Performance test 1: For this test, I will test the 10% selection rule of thumb. This rule of thumb states that if a selection on a table occurs, if less than 10% of the rows will be returned, it’s better to use an index scan. However once 10% selectivity is surpassed, it is typically more efficient to just do a full table scan. This is particularly true for unclustered indices, because the scan will bounce around to random pages, and will visit pages multiple times.

Scaled up versions of queries 2, 4, and 6 will be used in 3 different cases: one with no index on unique2, one with a clustered index on unique2, and one with a non-clustered index on unique1. The system parameters will remain set to default. This means all default query plan optimizations are enabled. Below 10% selectivity, we expect to see index scans perform faster than no index scans. Above 10% selectivity, we expect to see that a full file scan (no indices) performs faster than both index scans.

Here are the results:

|  |  |  |
| --- | --- | --- |
| 10% selection |  |  |
| Query 2 - clustered unique2 | Query 4 - no index unique2 | Query 6 - unclustered unique1 |
| 62 ms | 67 ms | 64 ms |
| 62 ms | 86 ms | 67 ms |
| 61 ms | 69 ms | 67 ms |
| 64 ms | 83 ms | 63 ms |
| 65 ms | 63 ms | 64 ms |
| result: 62.67 ms | result: 73 ms | result: 65 ms |
|  |  |  |
| 40% selection |  |  |
| Query 2 - clustered unique2 | Query 4 - no index unique2 | Query 6 - unclustered unique1 |
| 104 ms | 89 ms | 91 ms |
| 92 ms | 91 ms | 100 ms |
| 95 ms | 92 ms | 92 ms |
| 103 ms | 92 ms | 98 ms |
| 100 ms | 94 ms | 95 ms |
| result: 93.33 ms | result: 91.67 ms | result: 95 ms |

|  |  |  |
| --- | --- | --- |
| 2% selection |  |  |
| Query 2 - clustered unique2 | Query 4 - no index unique2 | Query 6 - unclustered unique1 |
| 57 ms | 67 ms | 57 ms |
| 57 ms | 128 ms | 57 ms |
| 50 ms | 56 ms | 55 ms |
| 55 ms | 65 ms | 58 ms |
| 65 ms | 59 ms | 54 ms |
| result: 56.33 ms | result: 62.67 ms | result: 56.33 ms |

These results don’t necessarily show that the 10% rule is exactly 10%, but it does demonstrate that the larger the selectivity percentage, the more likely a full table scan will outperform an index scan. In the 40% selectivity, the no index scan outperformed both clustered and unclustered scans. In the 2% selection, the clustered index scan roughly matches the unclustered. Once 10% is hit, the unclustered begins to take a bit longer than the clustered scan, but falls back to around the same during the 40% selection.

Performance test 2: For this test, I will be testing the various join algorithms that the optimizer chooses to use while disabling various join algorithms. A 5%, 15%, and 55% selectivity will be used. For each selectivity, *explain* will be used to determine which type of join the optimizer chose. Each join algorithm will be timed as before, then it will be disabled. This will force the query planner to use the next best algorithm it has chosen. I will be using these joins on 2 hundred thousand tuple relations.

I expect that as join algorithms are disabled, run time will increase, likely quite significantly. The piece I’m primarily interested in is the effect on the selectivity. I will also be showing the results for both the clustered and non-clustered indices.

Here are the results:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| hash join |  |  |  |  |  |
| 5% selectivity -  clustered | 5% selectivity -  unclustered | 15% selectivity -  clustered | 15% selectivity -  unclustered | 55% selectivity -  clustered | 55% selectivity -  unclustered |
| 92 ms | 105 ms | 116 ms | 124 ms | 192 ms | 277 ms |
| 100 ms | 105 ms | 114 ms | 122 ms | 194 ms | 275 ms |
| 86 ms | 94 ms | 113 ms | 125 ms | 196 ms | 294 ms |
| 96 ms | 92 ms | 113 ms | 133 ms | 198 ms | 268 ms |
| 88 ms | 101 ms | 117 ms | 140 ms | 254 ms | 278 ms |
| result: 91.67 ms | result: 100 ms | result: 114.33 ms | result: 127.33 ms | result: 196 ms | result: 276.66 ms |
|  |  |  |  |  |  |
| merge join |  |  |  |  |  |
| 5% selectivity -  clustered | 5% selectivity -  unclustered | 15% selectivity -  clustered | 15% selectivity -  unclustered | 55% selectivity -  clustered | 55% selectivity -  unclustered |
| 95 ms | 100 ms | 103 ms | 132 ms | 199 ms | 283 ms |
| 84 ms | 97 ms | 104 ms | 164 ms | 195 ms | 250 ms |
| 94 ms | 99 ms | 99 ms | 150 ms | 192 ms | 257 ms |
| 78 ms | 105 ms | 103 ms | 130 ms | 206 ms | 269 ms |
| 76 ms | 105 ms | 100 ms | 122 ms | 194 ms | 255 ms |
| result: 85.33 ms | result: 101.33 ms | result: 102 ms | result: 137.33 ms | result: 196 ms | result: 260.33 ms |
|  |  |  |  |  |  |
| nested loop join |  |  |  |  |  |
| 5% selectivity -  clustered | 5% selectivity -  unclustered | 15% selectivity -  clustered | 15% selectivity -  unclustered | 55% selectivity -  clustered | 55% selectivity -  unclustered |
| 96 ms | 101 ms | 126 ms | 131 ms | 240 ms | 291 ms |
| 99 ms | 103 ms | 130 ms | 128 ms | 230 ms | 252 ms |
| 102 ms | 101 ms | 120 ms | 131 ms | 280 ms | 292 ms |
| 101 ms | 100 ms | 124 ms | 129 ms | 214 ms | 269 ms |
| 95 ms | 108 ms | 120 ms | 141 ms | 240 ms | 221 ms |
| result: 98.66 ms | result: 101.66 ms | result: 123.33 ms | result: 130.33 ms | result: 236.66 ms | result: 270.66 ms |

These results are as expected, but there are a couple interesting tidbits. As the algorithms were disabled, results for 5% selectivity increased for both clustered and unclustered indices. For 15% selectivity, merge join actually outperformed the hash join by a fairly significant amount. At 55% selectivity, the various algorithms behaved rather similarly.

Performance test 3: For this test, I will be testing the differences between hot and cold hits across the various relation cardinalities. Since I have full control of this machine, I have the ability to restart the server process, which clears all buffer caches. I will be testing 2% and 55% selectivity, each using both the hundred thousand and million relation tables. Each test will also show no index search vs. clustered index search.

In particular, I will be using a join to create the results. This way the indices/no indices will play more of effect.

I expect the cold hits to take significantly more time the warm hits, however it will be interesting to see the differences in cold hits given a different selectivity. Also note that all default database settings will be retained.

Here are the results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2% Selectivity | cold clustered | cold no index | hot clustered | hot no index |
| 100,000 rel. | 140 ms | 232 ms | 107 ms | 79 ms |
|  | 221 ms | 141 ms | 77 ms | 85 ms |
|  | 181 ms | 165 ms | 73 ms | 76 ms |
| result: | **180.66 ms** | **179.3 ms** | **257 ms** | **80 ms** |
| 1,000,000 rel. | 338 ms |  | 259 ms |  |
|  | 293 ms |  | 261 ms |  |
|  | 273 ms |  | 226 ms |  |
| result: | **301.33 ms** |  | **248.66 ms** |  |
|  |  |  |  |  |
| 55% Selectivity | cold clustered | cold no index | hot clustered | hot no index |
| 100,000 rel. | 279 ms | 389 ms | 192 ms | 257 ms |
|  | 259 ms | 345 ms | 197 ms | 254 ms |
|  | 265 ms | 361 ms | 210 ms | 261 ms |
| result: | **267.66 ms** | **365 ms** | **199.66 ms** | **257.66 ms** |
| 1,000,000 rel. | 1.53 s |  | 1.52 s |  |
|  | 1.49 s |  | 1.48 s |  |
|  | 1.50 s |  | 1.47 s |  |
| result: | **1.51 s** |  | **1.49 s** |  |

As expected, the first load of a join query after a server restart is slower than subsequent runs. Rather surprisingly though, the timing saved is not very significant. I expected significant differences in the times, but they are much closer than I had expected. This could be due a couple factors. For example, it could be possible that there is still information being stored between server restarts. I did notice that after restarting the server that the sql db resides on, the first query of the day was slower than after just restarting postgres. Another possibility is the SSD could be making disk hits still rather cheap.

An interesting side note; the computer crashed during the unclustered search with a million relations. This could be due to computational limits, or the table itself could have issues with its values.

Performance test 4:

**Lessons learned:**