MariaDB System Research

BUFFER POOL

When we started up **MariaDB** and loaded the two 10K relations as well as the 1K relation, the buffer pool settings were: ~16K * 8192 pages = ~134M. The Wisconsin Benchmark states that "… the size of the benchmark relations should be at least a factor of 5 larger than the total main memory buffer space available. Thus, if the buffer pool is 4 megabytes, the benchmark relations should each contain 100,000 tuples. "
We'll use 2 identical relations with 1 million tuples each (each relation is ~204M), so to ensure optimal testing conditions, we set the buffer pool to be 40M.

The buffer pool has two lists, *old* and *new* which provide functionality akin to the *referenced* bit in the buffer pool we implemented in HW1. A new search result gets placed on the *old* list (maybe they could have chosen a different naming scheme), and the oldest drops off the list. If an *old* list member is accessed again, it's promoted to the top of the *new* list. The *new* list gets 63% of the buffer pool allotment and the *old* list gets 37%.

INDICES

MariaDB has several storage engine options, but the only one that seems relevant here is the default *InnoDB* storage engine. *InnoDB* uses *BTree* style indexing which allows for all equality operators, BETWEEN, and LIKE for searches beginning with a constant (but not for searches beginning with a wildcard).

JOIN ALGORITHMS

MariaDB has many options for Block Based Join Algorithms. These options are completely configurable, and the user can choose between regular (flat) and incremental, whether to use a hash or not, and whether to use an index or not:

- 1 – Flat BNL (Block Nested Loop)
- 2 – Incremental BNL
- 3 – Flat BNLH (Block Nested Loop Hash)
- 4 – Incremental BNLH
- 5 – Flat BKA (Block Key Access)
- 6 – Incremental BKA
- 7 – Flat BKAH (Block Key Access Hash)
- 8 – Incremental BKAH

The user can control these factors using the boolean variables *join_cache_incremental*, *join_cache_hashed*, and *join_cache_bka*. Additionally, the enumerated variable *join_cache_level* selects the set of permitted options from 0-8. No options above the selected level will be used. For example, if level 2 is selected, only Flat BNL or Incremental BNL will be permitted.

PostgreSQL System Research

INDICES

Postgres uses B-tree, Hash, GiST, SP-GiST, GIN, and BRIN indices.

- **B-tree:** A B-tree index is one laid out using a binary search tree (BST) data structure. Such a tree consists of a root node with two child nodes. The child to its left has a lesser value than the parent, while the child to the right has a greater value. Each child node also has two child nodes with the same parameters, and so on. Nodes that do not have children are called "leaf" nodes.
- **Hash:** A hash index is laid out on a hash table data structure. In this system, data is stored in an array of linear linked lists, where all data in that list has some aspect of its primary key in common (for example, when key % n equals the same number) that maps them to this specific spot (in the example above, array_index = key % n).
- **GiST:** A GiST, or Generalized Search Tree, is simply a modified B-tree index. In it, each node also contains a linear linked list of items that are equal to the first item on the chosen index variable.
- **SP-GiST:** SP-GiST is a form of GiST designed for use when your data naturally clusters around certain values.
- **GIN:** A GIN index, or Generalized Inverted Index, is an index created for searching words. It is structurally similar to a Hash tree. However, every index of the array represents a word, and every node in the accompanying LLL points to a piece of data that contains that word.
- **BRIN:** A BRIN, or Block Range Index, is ostensibly a 2-3-4-5 tree, a variant of the standard BST where each node can hold up to 4 items and have up to 5 children.

JOIN ALGORITHMS

Postgres supports Hash, Merge, and Nested Loop joins

- **Hash:** A hash join works by, unsurprisingly, creating a hash table. Each tuple in M is mapped to a unique index on the table. Then, every tuple of N that matches with an element in M via the join function is added to the LLL on that tuple.
- **Merge:** A sort-merge join operates by creating a new table, where each tuple is a joining, via the join function, of a tuple from M and a tuple from N.
- **Nested Loop:** A nested loop is very simple: for every tuple in M, search the entirety of N for tuples that match via the join function, and at each one of those, perform the join.

BUFFER POOL

Postgres actually has two default buffer pool sizes. The first is the size of the master buffer, which is 128MB. The second is the distributed buffer size, which is 8MB. The master buffer is the buffer pool for the whole SQL server. When a terminal accesses the server, a portion of

that master buffer of the distributed buffer size is set aside specifically for that particular interaction. Once the interaction ends, the buffer is rejoined with the master buffer. Both values can be modified in the postgresql.conf file.

QUERY TIMING

Postgres has two systems for measuring execution time: \timing and EXPLAIN ANALYZE. The first one, \timing, executes the query and tells you how long it took to execute, including time it spent printing the results and waiting to be scheduled in the processor. The second one, EXPLAIN ANALYZE, only times the query while it's in the processor. I could not find any information about how these two functions actually measure time.
(Source: https://www.postgresql.org/message-id/4D9EABDD.1060102@limsi.fr)

(Except as otherwise noted, all information comes from the official postgres documentation: https://www.postgresql.org/docs/)

Tests

PLAN

Any time we need an unclustered index, we'll create it on *unique1*, and when we need a clustered index, we'll created it on *unique2*.
In general, we'll create a temp table TMP to store the results of our queries to avoid large amounts of data printed to the screen.

We have 5 tables generated with our Python script (gendata.py) that we'll use for our tests:

- Two identical one million tuple relations, *onemtup1* and *onemtup2*.
- Two identical ten thousand tuple relations, *tenktup1* and *tenktup2*.
- One one thousand tuple relation, *onektup*.

JOIN

The metric we'll use to evaluate performance is the average elapsed time of 4 identical queries. Modified this from the WB queries to test a couple cases. The WB queries have been altered, but the query numbers left for reference.

One with 10% selection a small, right relation (similar to the WB Queries 9 and 12):

Query 9 (no index) and **Query 12** (clustered index) - JoinAselB (modified)

```
INSERT INTO TMP
SELECT * FROM onemtup1, tenktup1
WHERE (onemtup1.unique2 = tenktup1.unique2)
AND (tenktup1.unique2 < 1000)
```

Next with 2 large relations. From what we can tell, MariaDB does not support a Sort-Merge algorithm for Joins, which is ideal with 2 large relations. We're interested to see how the two systems differ (with the expectation that PostgreSQL chooses Sort-Merge):

Query 9 (no index) and **Query 12** (clustered index) - JoinAselB

```
INSERT INTO TMP
SELECT * FROM onemtup1, onemtup2
WHERE (onemtup1.unique2 = onemtup2.unique2)
AND (onemtup2.unique2 < 1000)
```

As far as expected results, we'd assume similar performance between the two systems for the first queries (1 million tuples joined with 10K tuples). We'd venture to guess that PostgreSQL will outperform MariaDB in the join of the large (1 million tuple) relations due to its use of the Sort-Merge algorithm.

UPDATE

The metric we'll use to evaluate performance is the average elapsed time of 10 identical queries.

The revised WB includes 2 queries to test the UPDATE command, each utilizing a single UPDATE. Even with a large relation, we'd expect a single UPDATE to run quickly, so we'll test 1000 UPDATES on using no index, updating a key attribute (with index), and updating a non-key attribute (with index).

Query 28 (no index) and **Query 31** (with index) - Update key attribute

```
UPDATE onemtup1
SET unique2 = -1
WHERE unique2 % 1000 = 0
```

Query 32 (with index) - Update indexed non-key attribute

```
UPDATE onemtup1
SET unique1 = -1
WHERE unique1 % 1000 = 0
```

We'd expect the no index UPDATES to take considerably longer than the indexed UPDATES, of course.

GROUP BY / AGGREGATE

These two sets of queries are meant to test two things. The first set is meant to test how the databases perform as the number of categories to be displayed increases and the number of tuples in each category decreases, and vice versa. The second set of queries is meant to determine how well they perform when asked to do different types of aggregation calculations.

The metric we will use for these queries is to run each one four times and time how long they take.

COUNT

- **Format:** Each of the queries will be in the form SELECT COUNT([data]) FROM [tuple] GROUP BY [data]
- [data] will be replaced with each of the following columns from the Benchmark schema:
    - two
    - four
    - ten
    - twenty
    - hundred
    - thousand
    - twothous (tenktup1 and tenktup2 only)
    - fivethous (tenktup1 and tenktup2 only)
    - tenthous (tenktup1 and tenktup2 only)
- [tuple] will be replaced with each of the following tables made using the Wisconsin schema:
    - onektup
    - tenktup1
    - tenktup2

AGGREGATE

- **Format:** Each of the queries will be in the form SELECT [AGGREGATE]([data]) FROM [tuple] GROUP BY [data]
- [tuple] will be replaced with each of the following tables made using the Wisconsin schema:
    - onektup
    - tenktup1
    - tenktup2
- [data] will be replaced with the datum for which each tuple performed the best in the first round of tests. For example, if the above queries were performed on tenktup1 and it was found that the GROUP BY hundred one had the shortest time, then all the AGGREGATE queries performed on tenktup1 will be grouped by hundred.
- [AGGREGATE] will be replaced with each of the following aggregate functions:
    - COUNT
    - SUM
    - MIN

- MAX
- AVG

Lessons Learned

CONFIGURATION

One challenge we both faced was in configuring the size of the buffer pool for each RDBMS. There are many config files and multiple config files where the documentation states the buffer pool may be adjusted, and it was necessary to restart the service, at least in the case of MariaDB, for the changes to take effect. The details on what ultimately worked required trial and error and it's still unclear why some config changes didn't work as expected. This tends to be the most common experience when dealing with configuration files.

BENCHMARK

Another challenge stemmed from the open-ended nature of the assignment. It was truly a challenge to attempt to come up with tests which would be interesting and worthwhile to perform in our RDBMS comparison. First, trying to understand the features of each system and considering which settings to tweak required a deep dive into the documentation for PostgreSQL and MariaDB. Next, benchmark design is an extremely geeky, back-end endeavor, and is not something you can easily Google for solutions. Finally, theorizing the results still seemed difficult, even after all our research, and we've yet to run the tests to see our theories were accurate.