Winter ‘21 DBMS Implementation (CS587)
Term Project Pt II
Benchmark Design

Aaron Hudson
Travis McGowan

System Research

BIGQUERY SYSTEM RESEARCH

BigQuery is, in many ways, a one size fits all system. As such there are few options for tuning query/database parameters. I find this to be not terribly surprising for a cloud Db product which essentially removes the DBA from managing anything to do with hardware. There are, however, a few options which do exist for optimizing or customizing query behavior and performance.

First, there is the processing location. This is entirely relevant as you are also able to select locations for tables/views/etc from Google’s various datacenters. Selecting an optimal location will mostly have to do with the location of the various tables involved.

There are also options (only two) for handling job priority: interactive vs batch. The system always defaults to interactive, even if you’ve saved it otherwise a thousand and one times. With ‘interactive‘ queries Google will identify an optimal batch size themselves and pool jobs to execute them together. If ‘batch’ is selected, queries will be executed discretely.

There is also the option of turning the cache preference on and off. This means that results from recent queries may be retained in memory for speeding the completion of later jobs.

A sproc equivalent is available via the ‘scheduling’ of queries in the BQ UI. Additionally, queries can be triggered by the client from the BQ API, further providing sproc equivalent functionality.

One of BigQuery’s most notable qualities is it’s lack of indices. Additionally, there are fairly heavy restrictions on the complexity of SQL allowed in the creation of materialized views. In the end, we were forced to create a materialized index, which was used to create a ‘current’ view for use in our tests.

CASSANDRA SYSTEM RESEARCH

Types of Indices

Cassandra allows creation of secondary indexes (in addition to the primary index, which correlates to the primary key of the table) that can be built for a column in a table. The secondary indexes are stored locally on each node in a hidden table and can be built in a background process.

Indexes in Cassandra are best used when a table has many rows that contain the indexed value. Indexes should be avoided on high-cardinality columns, tables that use a counter column, frequently updated/deleted columns, or to look for a row in a large partition unless narrowly queried.

Oftentimes it can be more efficient to use a materialized view for a specific query rather than relying on an index.

https://cassandra.apache.org/doc/latest/cql/indexes.html

https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/dml/dmlIndexInternals.html?hl=index

https://docs.datastax.com/en/archived/cql/3.3/cql/cql_using/useWhenIndex.html

Join Algorithms

Cassandra does not employ joins.

Buffer pool Size/Structure

32MB of 'file_cache_size_in_mb' configuration value is reserved for the buffer pool, with the remaining memory of that value used for chunk cache. There is also a flag 'buffer_pool_use_heap_if_exhausted' that is turned on by default that allows allocating memory on the heap if the buffer pool memory is exhausted.

The buffers can be allocated and managed with different options selected in the 'memtable_allocation_type' configuration value:

a. Heap_buffers = on heap nio buffers (default values)
b. Offheap_buffers = off heap (direct) nio buffers
c. Offheap_objects = off heap objects

Metrics for the buffer pool can be accessed via reports. 'Size' indicates the size of the managed buffer pool in bytes, and 'Misses' indicates the rate of misses in the pool (a higher value indicates more allocations required).

https://cassandra.apache.org/doc/latest/configuration/cassandra_config_file.html

https://cassandra.apache.org/doc/latest/operating/metrics.html

Measuring Query Execution Time

TRACING can be used to obtain information necessary to determine the time taken for a query to execute. When tracing is initialized, it results in a tracing session that saves tracing data for each query executed that can be accessed for up to 24 hours using the corresponding tracing_session_id. This data can also be exported if it needs to be saved for a longer period of time.

When tracing is running, it provides a table output after executing a query that contains a ‘source_elapsed’ column that contains the total microseconds elapsed for each part of the query process.

https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/cqlshTracing.html

POSTGRES SYSTEM RESEARCH

(There was some initial confusion, and we ended up completing this section for PostGres rather than the systems we chose to work on. We have included that work here anyway, because why not?)

1. enable_nestloop:

Is set to ‘on’ by default. Setting this to ‘off’ will force the DB to select ANY join method other than nested loop if it is available. That means that there is no way to turn it off completely. If there is no other method available, the database will still select a nested loop join.

There is a great deal of flexibility in making use of this method. It can be disabled for an individual user, the entire database, a DSN, or even a single query. However, the overwhelming majority of reputable advice instructs us not to use this method in production. The reason for this is that your database should not be erroneously selecting a nested join if it is tuned properly. In all likelihood the correct long term solution (if setting this method to ‘off’ *does* result in substantial time savings) is to refine the definitions of ‘random_page_cost’ or ‘seq_page_cost’in your settings.

https://community.microstrategy.com/s/article/Improve-Performance-Against-a-PostgreSQL-Database-by-Disabling-Nested-Loops?language=en_US

https://dba.stackexchange.com/questions/86100/what-are-the-side-effects-of-disabling-nested-loops-in-postgres

2a. seq_page_cost:

This setting has a default value of 1.0, and represents the database’s estimate of the relative cost of a sequential page I/O operation.

3a. random_page_cost

This database setting is defaulted to 4. This value denotes the conceptual cost of fetching a page of memory from the disk, non-sequentially.

2 & 3b. seq_page_cost & random_page_cost (as considered together)

Both of these values can be modified to match your particular server(s). The query planner uses these two costs, along with the relevant cpu costs, to calculate the expected cost of different plans and minimize run times. The appropriate value for these constants is determined by considering the specs of the server hard drive. The default values of 1 and 4 represent the most probable case, assuming the server uses traditional hard drives. In the case of using an SSD, the default values will definitely need to be changed as the data transfer rate is much higher. Another scenario to consider is that wherein the majority of the data is held in memory. In this case, it will usually make sense for seq_page_cost to be equal to random_page_cost.

“Hey! Don’t do *that…*”:

Both can be set to zero, but doing so will likely prove counterproductive, as it represents an impossible case. As a corollary, it makes no sense to set random_page_cost lower than seq_page_cost; though it is technically possible to do so. Lowering random_page_cost relative to seq_page_cost will lead the DB to favor index scans. Though I’m seeing advice that it is better to tune ‘effective_cache_size’ and ‘cpu_*’ costs if the goal is to make the DB favor indexes.

https://postgresqlco.nf/doc/en/param/random_page_cost/

https://dba.stackexchange.com/questions/204178/autodetect-random-page-cost-vs-seq-page-cost

4. shared_buffers

In addition to the buffer used by the kernel, Postgres uses an internal buffer space which is shared among its backend processes. This means that data is essentially stored in memory twice. First it is written to the internal buffer, then the kernel buffer, and finally to the disk.

The shared_buffer value (integer) specifies the amount of space to use for said ‘internal buffer’. Per *anybody’s* intuition, this value cannot be reset during production (The server must be reset after this has been updated.)

The default value for this parameter is 128 Mb, which is widely acknowledged as being far too low for nearly all applications. Postgres official documentation as well as the majority of technical resources indicate that the ideal range is near 25% of the total RAM of the server. Exceptions to this rule of thumb are for very small environments. In these cases the shared_buffer should skew lower than 25% with respect to the ‘smallness’ of the device memory. This is because Postgres operates on top of an OS, and space must be ceded to the OS kernel. Additionally, it sounds as though larger buffer sizes are less effective on Windows machines than other OS.

Big values for shared_buffer (40% and greater) are generally only beneficial in read heavy environments/use cases. Large shared_buffers are detrimental in write heavy environments, as the entire contents of shared_buffers are processed during writes. Large shared_buffer values generally require an update to the 'checkpoint_segments' value.

5. effective_cache_size:

This parameter is expressed as an integer, or direct expression of an amount of memory ie. 524288 -or- 4 Gb. (4 Gb being the default value of this parameter)

This parameter is used by the Postgres query planner to calculate the cost of indexes (in costsize.c). Higher values will lead Postgres to prefer index scans, while lower values lead Postgres to prefer sequential scans.

The value input by the dba is a representation of the expected memory available for disk caching. That is, the amount of memory left after accounting for the kernel buffer and shared_buffer (and some additional buffer for various other services). The recommended value is 50% of the device memory. On a dedicated server, a more common value is 75% of server memory.

Performance Experiment Design

For each experiment we will be comparing the following systems:

- BigQuery (w/o index)
- Cassandra (w/o index)
- Cassandra (w/ index)

1. Selectivity
    a. Experiment specifications
        i. This experiment is designed to test the performance of each database with queries of varying selectivities
        ii. Use a 1,000,000 tuple relation (scaled up version of TenKTup)
        iii. Modified from Wisconsin Benchmark Queries 1-6

```
SELECT *
FROM MMTup
WHERE unique2 BETWEEN selectivity_range;

selectivity_range:
        1% selectivity = 0 AND 9,999
        10% selectivity = 0 AND 99,999
        25% selectivity = 0 AND 249,999
        50% selectivity = 0 AND 499,999
```

75% selectivity = 0 AND 749,999

iv. n/a (option 1)

v. Expected results:

- At higher selectivities, BigQuery and Cassandra without an index will perform better
- At lower selectivities, Cassandra with an index will perform better

2. Scaling Data Size
    a. Experiment specifications
        i. This experiment is designed to test the performance of each database with varying relation sizes
        ii. Use the following relations (all scaled versions of TenKTup):
            - 1,000 tuples (OneKTup)
            - 10,000 tuples (TenKTup)
            - 100,000 tuples (CKTup)
            - 1,000,000 tuples (MMTup)
            - 10,000,000 tuples (MMKTup)
        iii. SELECT *
        FROM relation_to_test
        WHERE unique2 BETWEEN selectivity_range;

        selectivity_range:
        10% selectivity = 0 AND 99,999
        50% selectivity = 0 AND 499,999
        iv. n/a (option 1)
        v. Expected results:
            - As the relation size increases, BigQuery and Cassandra without an index will degrade faster
            - Cassandra with an index will remain stable

3. Bulk Update
    a. Experiment specifications
        i. This experiment is designed to test the performance of each database for performing updates to relations, and specifically will test varying amounts of data updated
        ii. Use a 1,000,000 tuple relation (scaled up version of TenKTup)
        iii. Modified from Wisconsin Benchmark Queries 28, 31, 32

```
UPDATE MMTup
SET unique2 = 1000001
WHERE unique2 = 145691;

UPDATE MMTup
SET unique2 = 1000001
WHERE unique2 BETWEEN 0 AND 99,999;
```

(will result in updating 100,000 tuples)

iv. n/a (option 1)

v. Expected results:

- BigQuery and Cassandra without an index will perform best for bulk updates that impact a lot of tuples, as they will not require maintenance of the index
- Cassandra with an index will perform best for limited updates that impact few tuples and can make use of the index to find the tuples to update

4. Views vs. No Views

a. Experiment specifications

i. This experiment is designed to test the performance impact using views have on each database. Specifically, the same query will be called on a relation using a view and without using a view. Furthermore, this test will be run on varying relation sizes to determine the impact of views as data sets grow.

ii. Use the following relations (all scaled versions of TenKTup):

- 100,000 tuples (CKTup)
- 1,000,000 tuples (MMTup)
- 10,000,000 tuples (MMKTup)

iii. Modeled after Wisconsin Benchmark Queries 1-10

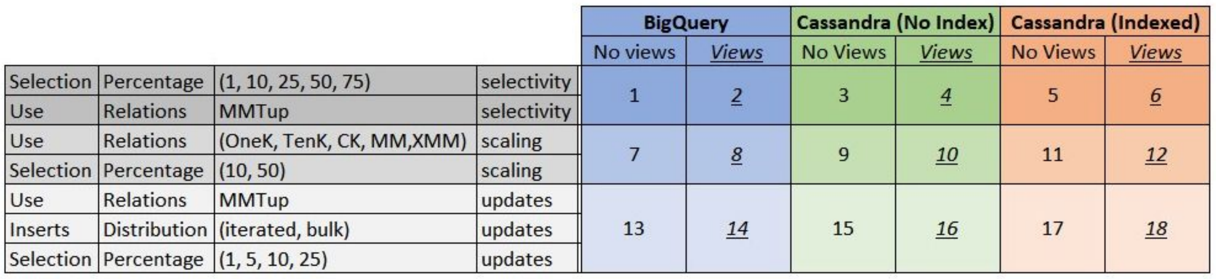
```
SELECT *
FROM relation_to_test
WHERE unique2 BETWEEN 0 AND 99;

SELECT *
FROM relation_to_test
WHERE unique2 < 50,000;
```

iv. n/a (option 1)

v. Expected results:

- Views will improve performance across the board, but will have greater impact as the relation size grows



				BigQuery		Cassandra (No Index)		Cassandra (Indexed)	
				No views	*Views*	No Views	*Views*	No Views	*Views*
Selection	Percentage	(1, 10, 25, 50, 75)	selectivity	1	*2*	3	*4*	5	*6*
Use	Relations	MMTup	selectivity						
Use	Relations	(OneK, TenK, CK, MM,XMM)	scaling	7	*8*	9	*10*	11	*12*
Selection	Percentage	(10, 50)	scaling						
Use	Relations	MMTup	updates	13	*14*	15	*16*	17	*18*
Inserts	Distribution	(iterated, bulk)	updates						
Selection	Percentage	(1, 5, 10, 25)	updates						

Figure 1: Breakdown of the various experimental tests, outlining overlapping test conditions.

Lessons Learned

Setting up Cassandra in GCP was fairly complicated, as it was our first time working with it. We ended up following a tutorial to get it started, and in the process learned a lot about Cassandra's layout. While Cassandra has a familiar design, there are some differences that required research on our part to get everything set up. Specifically of note is the difference in available data types for attributes, as the original implementation of TenKTup uses strings of a specific size, but string data types in Cassandra do not allow for that specificity. Also, the use of a keyspace feels a lot like a schema, but has some additional settings that we had to research.

Another area where we learned a lot was Cassandra budget management. Our initial setup resulted in a cost of about $4.80 a day just for the persistent storage (and nothing running). With the help of Neha, we were able to modify our instance to use cheaper settings that lowered the cost to about $0.12 a day. This was a nice opportunity to learn about the flexibility of GCP and what can be accomplished on a budget.

Additionally, we faced some challenges in the design of our experiments, as the two systems we are working with have some unique properties that limit the possible comparisons. Specifically, BigQuery doesn't use indices and Cassandra doesn't allow joins. This required us to think a little outside the box for different ways to compare them, as utilizing joins has been such a fundamental part of our database experience thus far.

Lastly, we spent quite a bit of time learning how to manage our data sets within GCP. This ranged from how to handle much larger sets of data, to pulling data from GCP storage into a specific instance, and then loading it into our relations. While it wasn't specifically challenging, it did require some reading of the documentation and time spent on our parts.

Overall, this portion of the project provided room to explore and develop familiarity with a lot of new technologies that will be beneficial for us in the future. GCP has been a challenging endeavor, but one that we know will worth the effort to learn.