

Benchmark Design

Part 2 Project submission

Shruti Nuchhi

I chose to implement both the options as I am interested to learn more in detail about Postgres, like how Postgres performs and scales in cases of complex queries, as a user what options we have to improve the performance. What are the advantages and drawbacks of using indices, when to use them and when not. What are the ways that Postgres would execute a given query, can it do any better than that given more flexibility? What is the point when things start getting worst, would like to identify and see can it be generalized to every kind of data or is it based on the table created (the number of columns and rows).

I am also interested to learn about cloud database and chose to see how Big Query performs. More interested to learn what advantage cloud has over the Postgres? What is the difference of doing things locally versus doing them on cloud? Would be interesting to see where Postgres would do better than Bigquery and vice versa.

Details of Big Query –

BigQuery stores data in the Capacitor columnar data format, and offers the standard database concepts of tables, partitions, columns, and rows. BigQuery uses Identity and Access Management (IAM) to manage access to resources. The three types of resources available in BigQuery are organizations, projects, and datasets. In the IAM policy hierarchy, datasets are child resources of projects. Tables and views are child resources of datasets — they inherit permissions from their parent dataset .A BigQuery slot is a unit of computational capacity required to execute SQL queries. BigQuery automatically calculates how many slots are required by each query, depending on query size and complexity.

Big Query's Query Optimization –

When evaluating query performance in BigQuery, the amount of work required depends on a number of factors such as

- a. Input data and data sources (I/O): How many bytes does the query read? The amount of data read by a query and the source of the data impact query performance and cost.
- b. Communication between nodes (shuffling): How many bytes does query pass to the next stage? How many bytes does query pass to each slot? - The amount of data that is shuffled directly impacts communication throughput and as a result, query performance. For example, a GROUP BY clause passes like values to the same slot for processing.
- c. Partitioning tables- A partitioned table is a special table that is divided into segments, called partitions that make it easier to manage and query data. By dividing a large table into smaller partitions, can improve query performance, and can control costs by reducing the number of bytes read by a query. There are two types of table partitioning that can be done in BigQuery:
 - a. Tables partitioned by ingestion time: Tables partitioned based on the data's ingestion (load) date or arrival date- When a table is created it can be partitioned on the ingestion time, Big Query automatically load data into daily, date- based partitions that reflects the data's ingestion or arrival date.

- b. Tables partitioned based on TIMESTAMP or DATE column - BigQuery also allows partitioned tables by TIMESTAMP or Date column. Data written to a partitioned table is automatically delivered to the appropriate partition based on the date value (expressed in UTC) in the partitioning column.
- d. Clustering Tables- We can create a clustered table in BigQuery only on the partitioned tables. The table data is automatically organized based on the contents of one or more columns in the table's schema. The columns that are specified are used to colocate related data. When we cluster a table using multiple columns, the order of columns specified is important. The order of the specified columns determines the sort order of the data. Clustering can improve the performance of certain types of queries such as queries that use filter clauses and queries that aggregate data. When data is written to a clustered table by a query job or a load job, BigQuery sorts the data using the values in the clustering columns. These values are used to organize the data into multiple blocks in BigQuery storage. When a query is submitted containing a clause that filters data based on the clustering columns, BigQuery uses the sorted blocks to eliminate scans of unnecessary data. Similarly, when you submit a query that aggregates data based on the values in the clustering columns, performance is improved because the sorted blocks colocate rows with similar values.

INDEXES and JOINS-

BigQuery executes a full-column scan every time it executes a query. BigQuery doesn't use or support indexes. Because BigQuery performance and query costs are based on the amount of data scanned during a query, the queries should be designed in such a way that the query refers only the columns that are relevant to the query.

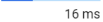
BigQuery supports the below kinds of JOIN. The syntax to use join is similar to that on POSTGRESQL. The join types supported are INNER, CROSS, ALL KINDS OF OUTER JOINS (FULL, LEFT, RIGHT).

QUERY PLAN EXPLANATION –

When BigQuery executes a query job, it converts the declarative SQL statement into a graph of execution, broken up into a series of query stages, which themselves are composed of more granular sets of execution steps. BigQuery leverages a heavily distributed parallel architecture to run these queries, and stages model the units of work that many potential workers may execute in parallel. Stages communicate with one another via fast distributed shuffle architecture.

Within the query plan, the terms of work units and workers are used, as the plan is conveying information specifically about parallelism. Top level job statistics provide the estimate of individual query cost using the total slot estimates of the query using this abstracted accounting. Another important property of the query execution architecture is that it is dynamic, which means that the query plan may be modified while a query is in flight. Stages that are introduced while a query is running are often used to improve data distribution throughout query workers. In query plans where this occurs, these are typically labeled as Repartition stages.

Elapsed time	Slot time consumed ?	Bytes shuffled ?	Bytes spilled to disk ?
0.1 sec	0.0 sec	175.78 KB	0 B ⓘ

Worker timing ⓘ						
Stages		Wait	Read	Compute	Write	Rows
✓ S00: Input ▾	Avg:	 1 ms	 50 ms	 6 ms	 37 ms	Input: 10,000
	Max:	 1 ms	 50 ms	 6 ms	 37 ms	Output: 1,000
✓ S02: Output ▲	Avg:	 2 ms	 6 ms	 16 ms	 4 ms	Input: 11,000
	Max:	 2 ms	 6 ms	 16 ms	 4 ms	Output: 1,000
READ <pre> \$1:unique1, \$2:unique2, \$3:two, \$4:four, \$5:ten, \$6:twenty, \$7:onepercent, \$8:tenpercent, \$9:twentypercent, \$10:fiftypercent, \$11:unique3, \$12:evenonepercent, \$13:oddonepercent, \$14:stringu1, \$15:stringu2, \$16:string4 FROM db-587-wisconsin-benchmark.Wisconsin_benchmark.TENKTUP1 WHERE less(\$2, 1000) </pre>						

POSTGRES PARAMETERS

I would like to explore few on the below mentioned Postgres parameters in particular.

1. **CLUSTER** - **CLUSTER** instructs PostgreSQL to cluster the table based on the index . The index must already have been defined on table. When a table is clustered, it is physically reordered based on the index information. Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered.
2. **enable_hashjoin** (boolean) – The hash join loads the candidate records from one side of the join into a hash table (marked with Hash in the plan) which is then probed for each record from the other side of the join. I would like to enable and disable the hash join to see how the queries on Postgres will perform with or without the hash joins. By default the option is enabled.
3. **work_mem** (integer) – The **work_mem** allows us to specify the amount of memory used by the internal sort options and hash tables before they are temporary disk files. The value is defaulted to 4MB. I plan to change the **work_mem** to a smaller value and run certain **ORDER BY** and **DISTINCT** projection queries to see how a small **work_mem** affects the performance. On a side note for a complex query, several sort or hash operations might be running in parallel; each operation will be allowed to use as much memory as this value specifies before it starts to write data into temporary files.
4. **shared_buffers**(integer)- Sets the amount of memory the database server uses for shared memory buffers. The default is typically 128 megabytes (128MB).This setting must be at least 128 kilobytes. (Non-default values of **BLCKSZ** change the minimum.) However, settings significantly higher than the minimum are usually needed for good performance. This parameter can only be set at server start. If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for **shared_buffers** is 25% of the memory in your system. There are some workloads where even large settings for **shared_buffers** are effective, but because PostgreSQL also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to **shared_buffers** will work better than a smaller amount.

Performance Experiment design

Queries planning to run –

1. Compare queries with and without indices and also with clustered and non clustered indices for Postgres.
Run the same queries with PARTITION for BigQuery.

- a. `SELECT * FROM TENKTUP1
WHERE unique2 BETWEEN 0 AND 99`
Run the above query using no index, clustered index and non clustered index for Postgres, and a normal as is query for Big Query. Compare the performance of all three circumstances. This query is run for 1% selection.

Expected results – I am expecting Postgres to perform better than BigQuery as the selection is less. Also I believe that the clustered index would not matter much in the execution of this query.

- b. `SELECT * FROM TENKTUP1
WHERE unique2 BETWEEN 792 AND 1791`
Run the above query using no index, clustered index and non clustered index for Postgres, and a normal as is query for Big Query. Compare the performance of all three circumstances. This query is run for 10% selection. If there is no difference seen in the execution time I would like to increase the selection criterion and run it for like 15% and more to see where the breakpoint actually occurs. Also would like to see how the Big Query and Postgres would perform for large selections.
Plan to use partition for BigQuery and compare it with index scan of Postgres.

Expected Results – I am expecting that BigQuery with partition will perform better than the Postgres without clustered index. I believe that Postgres would perform if clustered index is used.

2. Run queries to perform various join algorithms. Compare Postgres and BigQuery performance on joins. Also compare join using index and non index.

- a. `INSERT INTO TMP
SELECT * FROM TENKTUP1, TENKTUP2
WHERE (TENKTUP1.unique2 = TENKTUP2.unique2)
AND (TENKTUP2.unique2 < 1000)`

Run this and few other complex queries to perform certain joins. Plan to run this query with hash_join and without join hash join. I am expecting that when the hash_join is turned off the Postgres will use Merge Join. I will use the “set enable_hashjoin=off” to turn off the hash_join in Postgres as by default it is turned on. I wish to do all kinds of

join algorithms such as Cross Join, Inner join, Hash join (enable and disable), outer joins. Also I will perform some joins with and without index to see how the index helps during the joins.

Expected Results –

I am expected that the Postgres will do better with hash joins when compared to Big Query. In absence of hash joins I am guessing that Big query will do better than Postgres.

Also with index I believe the performance of Postgres is going to outperform all the our combinations.

3. Run different types of aggregate functions using Group by and having clauses. Compare Postgres with and without index and Bigquery with Postgres.

- a. `SELECT MIN (TENKTUP1.unique2) FROM TENKTUP1`
- b. `INSERT INTO TMP
SELECT MIN (TENKTUP1.unique3) FROM TENKTUP1
GROUP BY TENKTUP1.onePercent`
- c. `SELECT SUM (TENKTUP1.unique3) FROM TENKTUP1, TENKTUP2
WHERE TENKTUP1.unique2 = TENKTUP2.unique2
GROUP BY TENKTUP1.onePercent`

Expected Performance – Big Query's performance would be better than Postgres as the aggregate and group are slower in Postgres. I plan to do the group by with and without hash and without the index and expect that without index they are going to performed bad compared to with index and hashes.

4. Perform update and insert in both Postgres and Big Query and see how they perform. Plan to do a bulk update and do multiple inserts based on complex selection criteria. Update a key and a non key attribute. Also perform multiple updates on the field that is indexed and see how that performs as the index needs to be also updated accordingly every time there is an update to the indexed field.

- a. `INSERT INTO TENKTUP1 VALUES (10001, 74, 0, 2,0,10,50,688,
1950,4950,9950,1,100,
'MxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxGxxxxxxxxxxxxxxxxxxxxxxxxxC'
'GxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxCxxxxxxxxxxxxxxxxxxxxxxxxxA',
'OxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxOxxxxxxxxxxxxxxxxxxxxxxxxxO')`

- b. `Create INDEX TENKTUP1_INDEX ON TENKTUP1(unique1)`

```
UPDATE TENKTUP1
SET unique2 = 10001 WHERE unique2 = 1491
UPDATE TENKTUP1
SET unique1 = 10001 WHERE unique1 = 1491
```

Plan to write various updates to the unique1 field and see how this will perform.

Expected Performance – I am guessing that this is going to take longer time as the updates are made on the indexed field. Also this I plan to do only for Postgres.

Datasets -

I plan to use multiple datasets. I would use all the three tables that we created as part of the part one in our project and the run the above mentioned queries over these datasets to see how the performance changes with small to large datasets. I also plan to run the queries various time to average out the performance results. Basically would run the queries against ONEKTUP and TENKTUP1/TENKTUP2 to see the difference.

Lessons Learnt -

I learnt about the BigQuery and the optimization options available with BigQuery. Bigquery is column based database whereas Postgres is row-oriented database server. Also because of this I think that the Postgres will do better with insert and updates when compared to Bigquery.

With Bigquery when we doing joins we can't just do select * like we do in Postgres we would need to specify the table name like ONEKTUP.* incase the tables have the same column names.

Also BigQuery does not support indexes so we would have to be more careful when writing queries in BigQuery otherwise we would be unnecessarily increasing the overhead of the query output. Due to missing of the indexes I see that Postgres will do better in cases where indexes are used to get the query results.

BigQuery also does not have any support for Transaction based queries as BigQuery is solely designed to query data, which is currently it has no support for Transaction DB.

Also learnt that Postgres provides us with numerous options to increase the performance as per our requirement such as we can use work_mem to increase the work_mem so that large amount is pulled in and the queries are run faster. Also postgres provides us to turn on off various index options that are available which way we can actually understand the usage of each of the indexes are present. Postgres also provides with some of the Planner Cost Constants which by default are based on the cost of sequential page fetch.

I am yet to find a way to alter the join algorithms performed by Big Query.

References –

<https://cloud.google.com/bigquery/docs/parameterized-queries>

<https://www.postgresql.org/docs/9.4/runtime-config-resource.html>

<https://cloud.google.com/bigquery/query-plan-explanation>

Wisconsin benchmark paper and things discussed in class.