Danford Compton
Johnny Neckar
Aditya Sharoff

CS 487/587 Database Implementation
Winter 2021
Database Benchmarking Project - Part 2

Research:

Postgres:

Postgres is an open source, object-relational database management system that operates utilizing the SQL language (Postgresql.org 1). This project was started in 1986 by the University of California at Berkeley and has been actively developed for approximately 30 years (Postgresql.org 1). “PostgreSQL has earned a strong reputation for its proven architecture, reliability, data integrity, robust feature set, extensibility, and the dedication of the open source community behind the software to consistently deliver performant and innovative solutions” (Postgresql.org 1). This database is also ACID compliant which means it has atomicity, consistency, isolation, and durability. It also runs on every single operating system out there, which makes this tool incredibly useful for any data scientist/engineer.

Postgres supports a total of 6 different index types: B-tree, Hash, GiST, SP-GiST, GIN and BRIN. A B-tree index is the most commonly used of the 6, and is essentially just a data structure that contains a key value and a pointer to the next node. The leaf nodes of the B-tree contain offsets to the appropriate tuples. When a query is using a B-tree index it traverses down that tree to the appropriate leaf node and finds the tuple using the offset provided. A B-tree index can handle every type of comparison. A hash index occurs when the index you are hashing, such as student_id, is passed through a hash function and then mapped to a slot in an array containing buckets. The value passed to the hash function spreads the data among different buckets. This continues until the entire relation is sorted into the hash table. While this index may be extremely efficient, it only works for equality comparisons. The next index, known as GiST (Generalized Search Tree) is not an index so much as an infrastructure within which users can implement a multitude of different indexing strategies (postgres.org 1). Essentially, this is a tree that makes up for a shortcoming of the traditional B-tree index which only permits >=, <, =, >=, <=. Rather, GiSTs allow users to “define a rule to distribute data of an arbitrary type across a balanced tree and a method to use this representation for access by some operator” (Rogov 1). What this means is that you, as the user, are no longer constrained to using the traditional methods of comparison. Now you can use things such as &< to distribute your data which gives you greater flexibility when setting up your database. SP-GiST (SP stands for space partitioning) allows for users to set up non-balanced trees to allow users to customize their data structures (postgre.org 1). This is important because it allows for a level of customizability, which then allows users to craft appropriate structures tailored to their needs. GIN stands for Generalized Inverted Indexes, and is typically used when the items that are to be stored are objects such as arrays and text documents. BRIN indexes, or Block Range Indexes, are generally used when tables are extremely large, and can improve performance dramatically in these scenarios as opposed to the other methods listed.

There are three types of joins supported by Postgres: Nested Loop Join, Hash Join, and Merge Join. In a nested loop join, for every record in the outer relation, the entire inner relation is scanned to see if there is a match. This is incredibly inefficient and seldom used. A hash join is where you build a hash table according to the inner relation and you hash the outer relation to each inner relation tuple using a hash function. A Merge Join sorts the tables by the attribute that you want to join on and finds the matches.

In Postgres, the buffer pool is just an array that stores tables and indexes, with each slot in the buffer pool being able to store a single page (each slot is 8 KB). There are around 1000 such slots by default in the buffer pool which amounts to approximately 8000 KB of storage capacity in the buffer pool. The buffer pool is only a temporary location to store the various datafiles that are being read in from disk, so pages are written to and from this buffer pool as necessary. As such, Postgres uses the clock sweep algorithm to select the victim page which will be kicked out of the buffer pool. This clock sweep algorithm is a variant of the Not Frequently Used algorithm. In this algorithm, if a page's pin count is 0, it is the victim page. Otherwise, decrease the pin count by 1 and go forward. Wrap around and repeat this process until the victim slot is found.

The query execution time can be measured by adding EXPLAIN ANALYZE to the SELECT clause of a Postgres query (postgresql.org). When a user makes a query, Postgres generates a query plan for said query. By using the Explain clause, one is able to view the database's estimate with regards to how long a query will take to execute. Explain also gives details regarding the query plan that was generated. The ANALYZE EXPLAIN command tells both the estimated time for a query as well as how long it actually took. (postgresql.org)

Big Query:

BigQuery is the Google Cloud solution for when an organization or individual needs to query massive datasets. BigQuery is the public-accessible version of Dremel. Dremel is Google's internal big-data analysis system. It is capable of extremely fast speeds using cloud-based parallel processing. In Google's own words "Dremel Can Scan 35 Billion Rows Without an Index in Tens of Seconds" (Google) BigQuery has the distinct advantage of using a SQL language for database interaction. This means that even though it is very different under the hood, most SQL developers have no problem transitioning to the BigQuery service.

Due to this massive parallelization and the huge amount of storage available to Google, there is not a buffer pool in the normal sense. Instead the query becomes parallelized and Google leverages their fast internal speeds to quickly get your results without the typical buffer limitations.

In a similar fashion, BigQuery does not support any type of index. This is also due to the parallelization inherent in BigQuery. Google's reasoning for this is that using Dremel already makes it as fast or faster for most applications then having an index, and removing any indices saves space.

This also allows query execution time to be very precise as the result table is populated through their internal Jupiter network. Once this has completed, the end time is recorded, compared to the start time and given as output in the cloud shell.

BigQuery supports a number of different joins. These include Hash Joins, Broadcast Join, Self Joins, Cross Joins, and Skewed Joins. Self joins are when a table joins with itself. Google does not recommend using self joins as it is inefficient and analytic functions can produce the same results with less overhead. Hash Join is the same as described in the Postgres section. A cross join is when each tuple of one table is matched with every other tuple of another table. With a broadcast join the smaller table is sent to the larger table (this gets scaled up through parallelization) then the resultant table is stitched together as output. Skewed joins are another type that is not recommended and could be avoided by better data preprocessing. This occurs when the shuffling of data between tables is imbalanced, leading to processing hold-ups. This operation is still supported as there are some queries where it is unavoidable, but again, it is not recommended.

Due to BigQuery being a Google-specific product, the internals are a little opaque. It was not easy to find documentation that provides the detail that we were interested in. As such, this section is more vague than what we think is ideal but hopefully we will discover some answers in our experimentation.

Sources:
https://medium.com/Postgres-professional/indexes-in-Postgresql-5-gist-86e19781b5db
https://www.Postgresql.org/docs/9.5/gin-intro.html
https://www.Postgresql.org/docs/12/spgist-intro.html#:~:text=SP%2DGiST%20is%20an%20abbreviation,and%20radix%20trees%20(tries).&text=These%20popular%20data%20structures%20were%20originally%20developed%20for%20in%2Dmemory%20usage.
https://medium.com/Postgres-professional/indexes-in-Postgresql-5-gist-86e19781b5db

https://www.mssqltips.com/sqlservertip/3099/understanding-sql-server-memoryoptimized-tables-hash-indexes/#:~:text=Basically%2C%20a%20hash%20index%20is,bucket%20of%20the%20hash%20index.
https://database.guide/what-is-acid-in-databases/#:~:text=In%20database%20systems%2C%20ACID%20(Atomicity,database%20transactions%20are%20processed%20reliably.&text=An%20ACID%2Dcompliant%20DBMS%20ensures,consistent%20despite%20any%20such%20failures
https://www.postgresql.org/docs/9.5/indexes-types.html
https://cloud.google.com/files/BigQueryTechnicalWP.pdf
https://www.postgresql.org/docs/9.1/sql-explain.html

Experiment Design:

Note: We are doing option two, so we are excluding subsection ‘d’ because it is not applicable to our work.

1. Loading into a database.
    a. This test explores the loading time into both databases. We will test between loading from Google Drive vs Google Buckets
    b. We will use the 100,000 tuple set as well as creating a 500,000 tuple set.
    c. This will be testing load times. We will run a query after loading to observe the database behavior, such as query 5.
    d. From our previous attempts at loading, it would seem that loading from drive does not necessarily mean that BigQuery keeps the database. It seemed more that it was loading from Drive every time, so we will try to reproduce that behavior and compare it to loading from a Google Bucket. It also might be the case that the data needs to ‘settle’ in BigQuery, so we will repeat enough times to come to some sort of conclusion.

       Query 5 - 1% selection via a non-clustered index INSERT INTO TMP SELECT * FROM TENKTUP1 WHERE unique1 BETWEEN 0 AND 99

2. Projection differences between databases.
    a. This test will explore the performance differences between the two systems in regards to projection.
    b. We will use the 10,000 tuple set as well as the 100,000 tuple set. If the results are not appreciably different we may move up to the 500,000 tuple set that we create for experiment 1.
    c. This will use queries 18 and 19.

   d. We suspect that Postgres will take a longer time for the larger tuple sets, and that BigQuery will have more consistent performance regardless of the set size.

      Query 18 - Projection with 1% Projection
      INSERT INTO TMP
      SELECT DISTINCT two, four, ten, twenty, onePercent, string4
      FROM TENKTUP1

      Query 19 - Projection with 100% Projection
      INSERT INTO TMP
      SELECT DISTINCT two, four, ten, twenty, onePercent,
      tenPercent, twentyPercent, fiftyPercent, unique3,
      evenOnePercent, oddOnePercent, stringu1, stringu2, string4
      FROM TENKTUP1

3. Joins (It's a good experiment)
   a. This will explore how the systems respond to joins differently.
   b. This will use the 1k, 10k, 100k, and maybe 500k tuple sets.
   c. We will use queries 9, 10, and 11 because those are the join tests.
   d. Postgres will most likely take an increasing amount of time to complete the joins as the size of the set increases. BigQuery is the big question, because, according to some of our research, BigQuery does not like joins. So we suspect that this will be significantly 'harder' for BigQuery

      Query 9 (no index) - JoinAselB INSERT INTO TMP SELECT * FROM TENKTUP1, TENKTUP2 WHERE (TENKTUP1.unique2 = TENKTUP2.unique2) AND (TENKTUP2.unique2 < 1000)

      Query 10 (no index) - JoinABprime INSERT INTO TMP SELECT * FROM TENKTUP1, BPRIME WHERE (TENKTUP1.unique2 = BPRIME.unique2)

      Query 11 (no index) - JoinCselAselB INSERT INTO TMP SELECT * FROM ONEKTUP, TENKTUP1 WHERE (ONEKTUP.unique2 = TENKTUP1.unique2) AND (TENKTUP1.unique2 = TENKTUP2.unique2) AND (TENKTUP1.unique2 < 1000)

4. Run Speed Variation

a. This experiment will test the variation between run speeds (without cashing) due to the cloud environment. We will run the same, difficult tests over and over again to see what kind of variation occurs.
b. We will use the 100k and 500k tuple sets to get the highest times possible.
c. We will use queries 11, 15 and 19 due to their complexity along with some variation. (this is subject to change for more variety)
d. We think that due to the data transfer complexities that occur with cloud systems that the results should end up mostly within 1-2% of themselves, but we are expecting wild outliers.

   Query 11 (no index) - JoinCselAselB INSERT INTO TMP SELECT * FROM ONEKTUP, TENKTUP1 WHERE (ONEKTUP.unique2 = TENKTUP1.unique2) AND (TENKTUP1.unique2 = TENKTUP2.unique2) AND (TENKTUP1.unique2 < 1000)

   Query 15 (non-clustered index) - JoinAselB INSERT INTO TMP SELECT * FROM TENKTUP1, TENKTUP2 WHERE (TENKTUP1.unique1 = TENKTUP2.unique1) AND (TENKTUP1.unique2 < 1000)

   Query 19 - Projection with 100% Projection INSERT INTO TMP SELECT DISTINCT two, four, ten, twenty, onePercent, tenPercent, twentyPercent, fiftyPercent, unique3, evenOnePercent, oddOnePercent, stringu1, stringu2, string4 FROM TENKTUP1

5. Math execution speed
   a. This experiment will test the execution time of aggregate and sum queries.
   b. We will use 10k, 100k, and 500k tuple sets.
   c. This experiment will use queries 20, 21, and 22.
   d. We anticipate that this will be quick with gradually increasing times for Postgres, but work like lightning with BigQuery.

      Query 20 (no index) Minimum Aggregate Function INSERT INTO TMP SELECT MIN (TENKTUP1.unique2) FROM TENKTUP1

      Query 21 (no index) Minimum Aggregate Function with 100 Partitions INSERT INTO TMP SELECT MIN (TENKTUP1.unique3) FROM TENKTUP1 GROUP BY TENKTUP1.onePercent

Query 22 (no index) Sun Aggregate Function with 100 Partitions INSERT INTO TMP SELECT SUM (TENKTUP1.unique3) FROM TENKTUP1 GROUP BY TENKTUP1.onePercent

What We Learned:

In this project we learned how to set up Big Query on google cloud. We initially tried to use BigQuery through the command line interface but after struggling to get the database up and running, we found it much more usable through the GCP console dashboard. We did run into some strange unexplained behavior when using BigQuery, as some queries we ran had a significant delay before being executed. Our hypothesis is that this may be related to the storage method of the data being accessed (eg gdrive vs GCP bucket), which informed the design on our first experiment.

We also gained insight into the machinations of both big query and postgres. Some of the things we learned included the types of indices used in the two systems (in this case postgres because Big Query doesn't use indices), the join algorithms used, buffer pool structure/size, general facts, and how to time the queries. With this knowledge, we are equipped to set up more efficient databases, and test these systems more thoroughly than before.

What we found to be especially interesting is the non-Google sources that claim that BigQuery doesn't like joins. This is not something that we had considered and we took the example join experiment because of exactly this.