Data Warehouse Guide

A Deep Dive Into Google BigQuery Architecture

Understand how BigQuery works and comes in handy when optimizing for query performance and high cost effectiveness.
Google's BigQuery is an enterprise-grade cloud-native data warehouse. BigQuery was first launched as a service in 2010 with general availability in November 2011. Since inception, BigQuery has evolved into a more economical and fully-managed data warehouse which can run blazing fast interactive and ad-hoc queries on datasets of petabyte-scale. In addition, BigQuery now integrates with a variety of Google Cloud Platform (GCP) services and third-party tools which makes it more useful.

BigQuery is serverless, or more precisely data warehouse as a service. There are no servers to manage or database software to install. BigQuery service manages underlying software as well as infrastructure including scalability and high-availability. The pricing model is quite simple - for every 1 TB of data processed you pay $5. BigQuery exposes simple client interface which enables users to run interactive queries.

Overall, you don't need to know much about underlying BigQuery architecture or how this service operates under the hood. That's the whole idea of BigQuery - you don't need to worry about architecture and operation. To get started with BigQuery, your must be able to import your data into BigQuery, then be able to write your queries using SQL dialects offered by BigQuery.

Having said that, a good understanding of BigQuery architecture is useful when implementing various BigQuery best-practices including controlling costs, optimizing query performance, and optimizing storage. For instance, for best query performance, it is highly beneficial to understand how BigQuery allocates resources and relationship between the number of slots and query performance.

High-level architecture

BigQuery is built on top of Dremel technology which has been in production internally in Google since 2006. Dremel is Google's interactive ad-hoc query system for analysis of read-only nested data. Original Dremel papers were published in 2010 and at the time of publication Google was running multiple instances of Dremel ranging from tens to thousands of nodes.

10,000 foot view

BigQuery and Dremel share the same underlying architecture. By incorporating columnar storage and tree architecture of Dremel, BigQuery offers unprecedented performance. But, BigQuery is much more than Dremel. Dremel is just an execution engine for the BigQuery. In fact, BigQuery service leverages Google's innovative technologies like Borg, Colossus, Capacitor, and Jupiter. As illustrated below, a BigQuery client (typically BigQuery Web UI or bg command-line tool or REST APIs) interact with Dremel engine via a client interface. Borg - Google's large-scale cluster management system - allocates the compute capacity for the Dremel jobs. Dremel jobs read data from Google's Colossus file systems using Jupiter network, perform various SQL operations and return results to the client. Dremel implements a multi-level serving tree to execute queries which are covered in more detail in following sections.

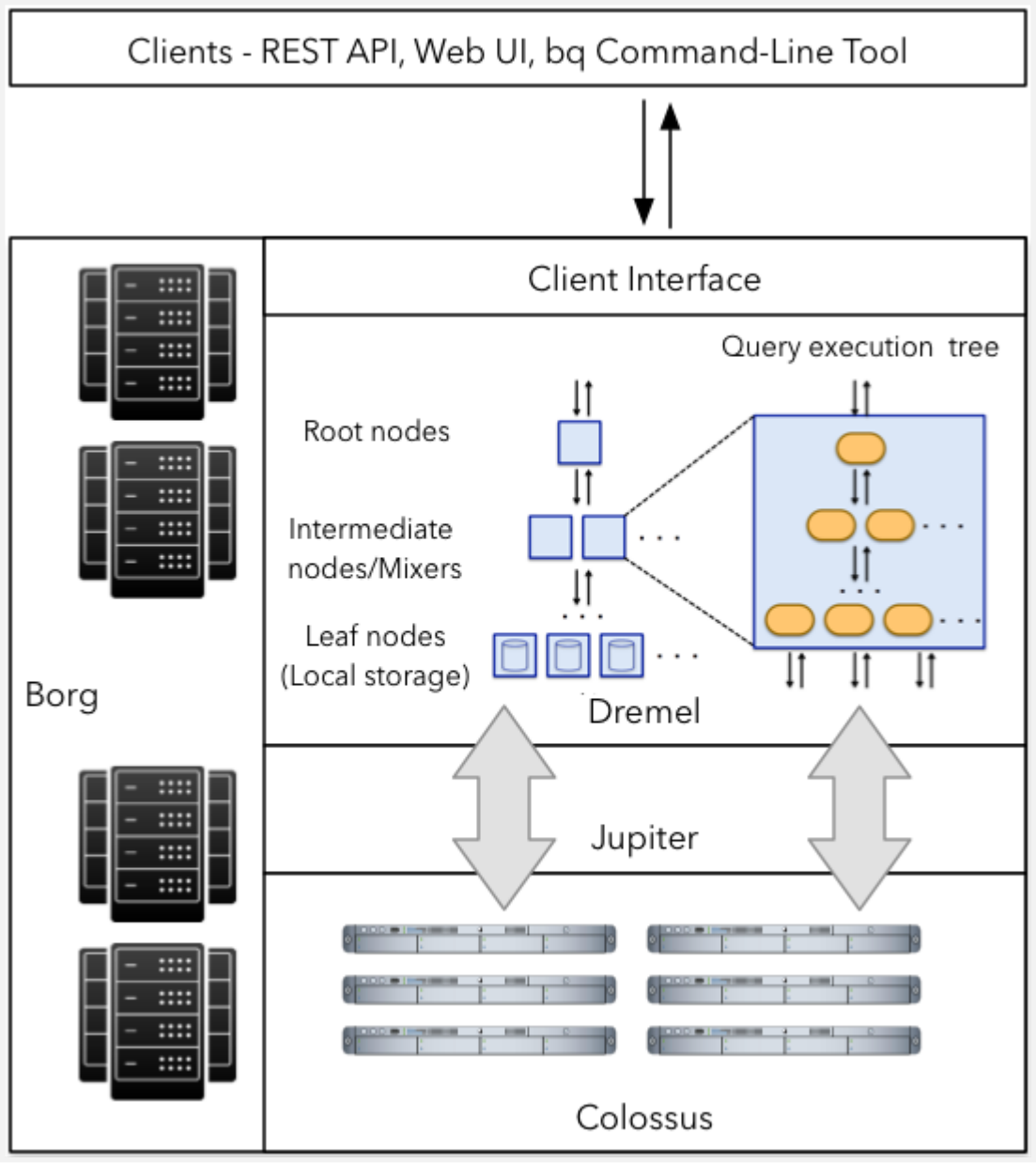

Figure-1: A high-level architecture for BigQuery service.

It is important to note, BigQuery architecture separates the concepts of storage (Colossus) and compute (Borg) and allows them to scale independently - a key requirement for an elastic data warehouse. This makes BigQuery more economical and scalable compared to its counterparts.

Storage

The most expensive part of any Big Data analytics platform is almost always disk I/O. BigQuery stores data in a columnar format known as Capacitor. As you may expect, each field of BigQuery table i.e. column is stored in a separate Capacitor file which enables BigQuery to achieve very high compression ratio and scan throughput. In 2016, Capacitor replaced ColumnIO - the previous generation optimized columnar storage format. Unlike ColumnIO, Capacitor enabled BigQuery to directly operate on compressed data, without decompressing the data on the fly.

You can import your data into BigQuery storage via Batch loads or Streaming. During the import process, BigQuery encodes every column separately into Capacitor format. Once all column data is encoded, it's written back to Colossus. During encoding various statistics about the data is collected which is later used for query planning.

BigQuery leverages Capacitor to store data in Colossus. Colossus is Google's latest generation distributed file system and successor to GFS (Google File Systems). Colossus handles cluster-wide replication, recovery and distributed management. It provides client-driven replication and encoding. When writing data to Colossus, BigQuery makes some decision about initial sharding strategy which evolves based on the query and access patterns. Once data is written, to enable the highest availability BigQuery initiates geo-replication of data across different data centers.

In a nutshell, Capacitor and Colossus are key ingredients of industry-leading performance characteristics offered by BigQuery. Colossus allows splitting of the data into multiple partitions to enable blazing fast parallel read whereas Capacitor reduces requires scan throughput. Together they make possible to process a terabyte data per second.

Native vs. external

So far we have discussed the storage for the native BigQuery table. BigQuery can also perform queries against external data sources without the need to import data into the native BigQuery tables. Currently, BigQuery can perform direct queries against Google Cloud Bigtable, Google Cloud Storage, and Google Drive.

When using an external data source (aka federated data source), BigQuery performs on-the-fly loading of data into Dremel engine. Generally speaking, queries running against external data

sources will be slower than native BigQuery tables. Performance of queries also depends on external storage type. For instance, queries against Google Cloud Storage will perform better than Google Drive. If performance is a concern then you should always import your data into BigQuery table before running the queries.

Compute

BigQuery takes advantage of Borg for data processing. Borg simultaneously runs thousands of Dremel jobs across one or more clusters made up of tens of thousands of machines. In addition to assigning compute capacity for Dremel jobs, Borg handles fault-tolerance.

Network

Apart from disk I/O, big data workloads are often rate-limited by network throughput. Due to the separation between compute and storage layers, BigQuery requires an ultra-fast network which can deliver terabytes of data in seconds directly from storage into compute for running Dremel jobs. Google's Jupiter network enables BigQuery service to utilize 1 Petabit/sec of total bisection bandwidth.

Read-only

Due to columnar storage, existing records cannot be updated hence BigQuery primarily supports read-only use-cases. That said, you can always write the processed data back into new tables.

Execution model

Dremel engine uses a multi-level serving tree for scaling out SQL queries. This tree architecture has been specifically designed to run on commodity hardware. Dremel uses a query dispatcher which not only provides fault tolerance but also schedules queries based on priorities and the load.

In a serving tree, a root server receives incoming queries from clients and routes the queries to the next level. The root server is responsible to return query results to the client. Leaf nodes of the serving tree do the heavy lifting of reading the data from Colossus and performing filters and partial aggregation. To parallelize the query, each serving level performs (Root and Mixers) query rewrite and ultimately modified and partitioned queries reach to the leaf nodes for execution.

During query rewrite, few things happen. Firstly, the query is modified to include horizontal partitions of the table, i.e. shards (in original Dremel paper shards were referred as tablets). Secondly, certain SQL clause can be stripped out before sending to leaf nodes. In a typical Dremel tree, there are hundreds or thousands of leaf nodes. Leaf nodes return results to Mixers or intermediate nodes. Mixers perform aggregation of results returned by leaf nodes.

Each leaf node provides execution thread or number of processing units often called as slots. It is important to note, BigQuery automatically calculates how many slots should be assigned to each query. The number of allocated slots depending on query size and complexity. At the time of writing of this article, for on-demand pricing model maximum 2000 concurrent slots are allowed per BigQuery project.

To help you understand how Dremel engine works and how serving tree executes, let's look into a simple query,

```
SELECT A, COUNT(B) FROM T GROUP BY A
```

When root server receives this query, the first thing it does is translate the query into a form which can be handled by next level of serving tree. It determines all shards of table T and then simplifies the query.

```
SELECT A, SUM(c) FROM (R1i UNION ALL ... R1n ) GROUP BY A
```

In this case R11, R12, . . . , R1n are results of queries sent to the Mixer 1, . . . , n at level 1 of the serving tree.

Next Mixers modify the incoming queries so that they can pass it to Leaf nodes. Leaf nodes receive the customized queries and read data from Colossus shards. A Lead node reads data for columns or fields mentioned in the query. As leaf node scans the shards, it walks through the opened column files in parallel, one row at a time.

Depending on the queries, data may be shuffled between leaf nodes. For instance, when you use GROUP EACH BY in your queries, Dremel engine will perform shuffle operation. It is important to understand the amount of shuffling required by your queries. Some query with operations like JOIN can run slow unless you optimise them to reduce the shuffling. That's why BigQuery recommends trimming the data as early in the query as possible so that shuffling due to your operations is applied to a limited data set.

As BigQuery charges you for every 1 TB of data scanned by leaf nodes, we should avoid scanning too much or too frequently. There are many ways to do this. One is partitioning your tables by date. For each table, additional sharding of data performed by BigQuery which you can't influence. Instead of using one big query, break them into small steps and for each step save query results into intermediate tables so that subsequent queries have less data scan. It may sound counter-intuitive but the LIMIT clause does not reduce the amount of data get scanned by a query. If you just need sample data for exploration, you should use Preview options and not a query with the LIMIT clause.

At each level of serving tree, various optimisations are applied so that nodes can return results as soon as they ready to be served. This includes tricks like priority queue or streaming results.

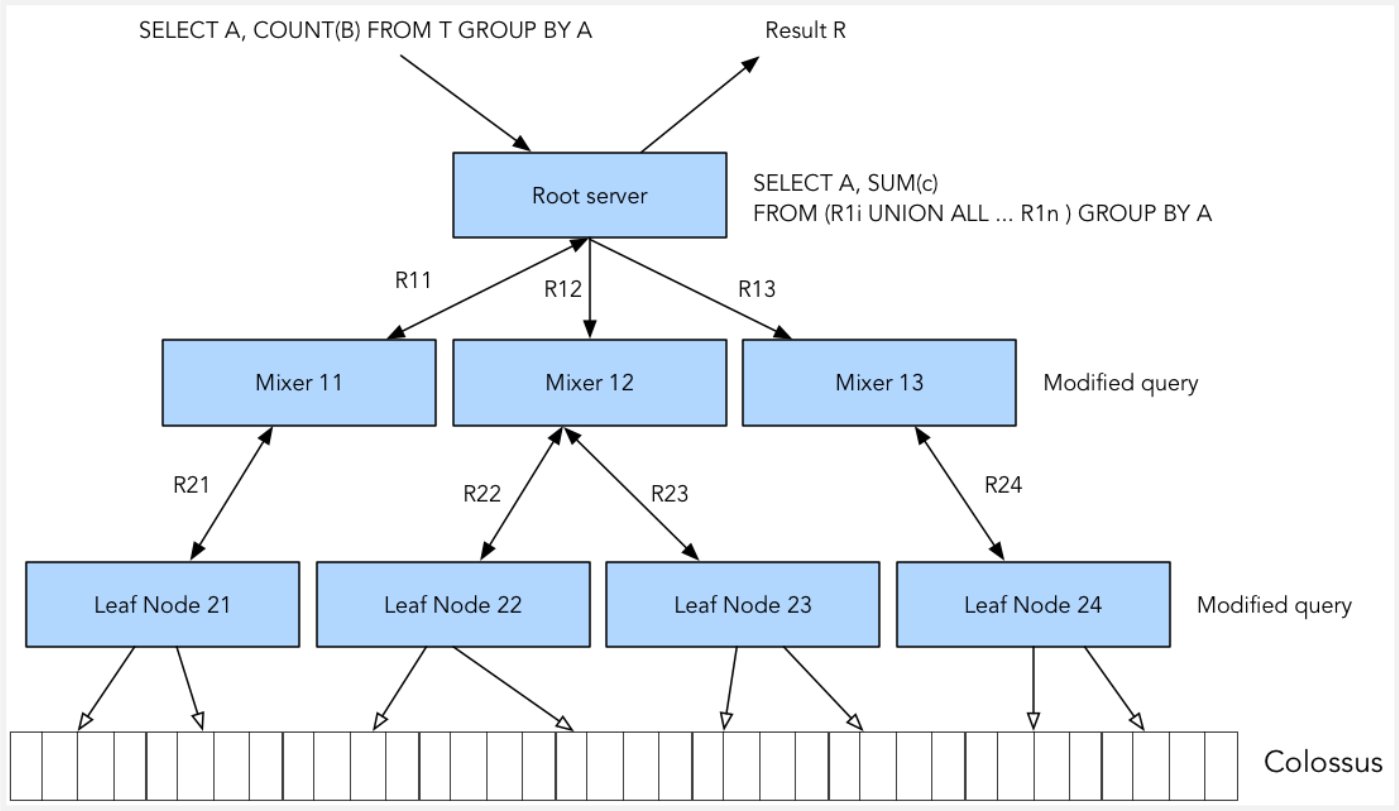

Figure-2: An example of Dremel serving tree.

Some mathematics

Now that we understand BigQuery architecture, let's look into how resources allocation played out when you run an interactive query using BigQuery. Say you are querying against a table of 10 columns with storage 10TB and 1000 shards. As discussed earlier, when you import your data into a table BigQuery service determines the optimal number of shards for your table and tunes it based on data access and query pattern. During data import, BigQuery will create Capacitor files - one for each column of the table. In this particular case, 10 Capacitor files per shard.

Effectively, there are 1000 x 10 files to read if you perform a full scan (i.e. select * from table). To read these 10000 files you have 2000 concurrent slots (if you are on BigQuery on-demand pricing model and assuming this is only interactive query you are running under your BigQuery project), so on average, one slot will be reading 5 Capacitor files or 5 GB of data. To read this

much data using Jupiter network it will take anywhere ~4 seconds (10 Gbps) which is one of the key differentiators for BigQuery as a service.

Obviously, we should avoid full scan because a full scan is most expensive - both computationally as well as cost wise - way to query your data. We should query only the columns that we need and that's an important best-practice for any column-oriented database or data warehouse.

Data model

BigQuery stores data as nested relations. The schema for a relation is represented by a tree. Nodes of the tree are attributes, and leaf attributes hold values. BigQuery data is stored in columns (leaf attributes). In addition to compressed column values, every column also stores structure information to indicate how the values in a column are distributed throughout the tree using two parameters - definition and repetition levels. These parameters help to reconstruct the full or partial representation of the record by reading only requested columns.

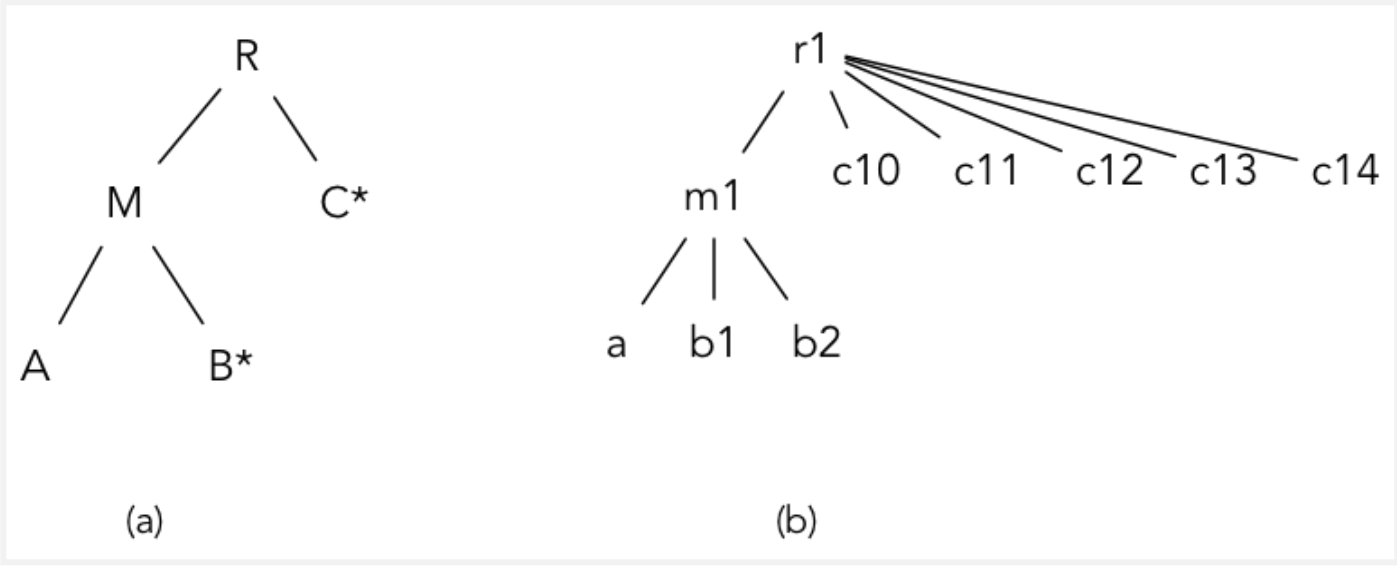

Figure-3: An example of (a) tree schema and (b) an instance based on the schema.

To take full advantage of nested and repeated fields offered by BigQuery data structure, we should denormalize data whenever possible. Denormalization localizes the necessary data to

individual nodes which reduce the network communication required for shuffling between slots.

Query language

BigQuery currently supports two different SQL dialects: standard SQL and legacy SQL. Standard SQL is compliant with the SQL 2011 and offers several advantages over the legacy alternative. Legacy SQL is original Dremel dialect. Both SQL dialects supports user-defined functions (UDFs). You can write queries in any format but Google recommends standard SQL. In 2014, Google published another paper describing how Dremel engine uses a semi-flattening tree data structure along with a standard SQL evaluation algorithm to support standard SQL query computation. This semi-flattening data structure is more aligned the way Dremel processes data and is usually much more compact than flattened data.

It is important to note that when denormalizing your data you can preserve some relationships taking advantage of nested and repeated fields instead of completely flattening your data. This is exactly what we will call semi-flattening data structure.

Alternatives

Although there are several alternatives of BigQuery - both in the open-source domain and cloud-based as service offerings - it still remains difficult to replicate the scale and performance of BigQuery. Primarily because Google does a fantastic job in blending infrastructure with BigQuery software. Open source solutions such as Apache Drill and Presto require a massive infrastructure engineering and ongoing operational overhead to match the performance of BigQuery. Amazon Athena - a serverless interactive query service offered by Amazon Web Services (AWS) - is hosted version of Presto with ANSI SQL support but this service is relatively new. Initial benchmarks suggest that the BigQuery still has a massive edge in terms of performance.

Final thoughts

BigQuery is designed to query structured and semi-structured data using standard SQL. It is highly optimized for query performance and provides extremely high cost effectiveness. BigQuery is a cloud-based fully-managed service which means there is no operational overhead. It is more suitable for interactive queries and OLAP/BI use cases. Google's cloud infrastructure technologies such as Borg, Colossus, and Jupiter are key differentiator why BigQuery service outshines some of its counterparts.

References

1. Dremel: Interactive Analysis of Web-Scale Datasets
2. Large-scale cluster management at Google with Borg
3. Storing and Querying Tree-Structured Records in Dremel
4. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Data center Network

Built by Panoply