A Survey of Google Cloud BigTable

Jitindeep Dhillon
Concordia University
Montreal, Canada
legitdhillon@gmail.com

Vatsal Chauhan
Concordia University
Montreal, Canada
vatsal91098@gmail.com

Faddilin Mahady Riniyad
Concordia University
Montreal, Canada
riniyad@gmail.com

Harshil Patel
Concordia University
Montreal, Canada
harshilpatel1903@gmail.com

ABSTRACT

We performed a survey on Google's BigTable. Taking into consideration the distributed nature of BigTable, we have analyzed the infrastructure on which it is build, the key features it offers as a distributed system, and an analysis of how BigTable performs in reading and writing data.

PVLDB Reference Format:
Jitindeep Dhillon, Vatsal Chauhan, Faddilin Mahady Riniyad, and Harshil Patel. A Survey of Google Cloud BigTable. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at http://vldb.org/pvldb/format_vol14.html.

1 INTRODUCTION

Bigtable began as the storage system for the web search index at Google and has become one of the main technologies backing many of their other storage systems such as Megastore and Cloud Datastore [5] . It was built to solve a specific but complex problem: updating petabytes of data, with incredibly high throughput, low latency, and high availability.

A traditional approach of using a relational SQL database does not come close to solving this problem. SQL databases fall over quickly due to drastic increase in query response time when the database is scaled horizontally. Two of the major features of relational databases, transactions and joins, don't scale very well when operations are performed across different nodes. Waiting for operations to complete across nodes or transferring data across nodes takes time, which slows down the entire process. (Neo4j, 2015). Google instead came up with a highly performant usage of a globally sorted key-value map, which automatically rebalances data based on service usage to reach the performance and scale requirements needed [5].

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

2 INFRASTRUCTURE CONCEPTS

Bigtable acts as a managed service, which means that management of individual virtual machines is unnecessary; unlike a HBase cluster. In the hierarchy of Bigtable there are a tree of components that help form the system and provide the speed and functionality for an instance. This section briefly describes these components and how they relate to one another.

2.1 Storage Model

The storage model of Bigtable is much more like a jagged key-value map than a grid. In fact, researchers from Google describe Bigtable as "a sparse, distributed, persistent, multi-dimensional sorted map" [5].

In short, Bigtable is a big key-value store that distributes data across many servers while keeping keys in that map sorted. Thanks to that global sorting, Bigtable allows both key lookups (as in any key-value store) as well as scans over key ranges and key prefixes.

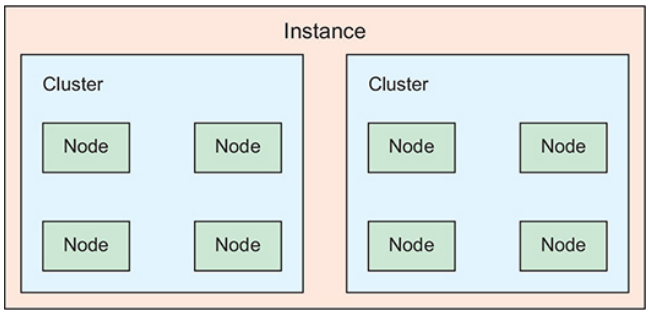

Figure 1: Hierarchy of instances, clusters, and nodes 4 [6]

2.2 Instance

At the top level is the Bigtable instance, which represents the container for your data and acts as the service for which your application relies on. Instance's have a few options that you can configure, such as the medium on which the data is stored (HDD or SSD) and application profiles [4]. Application profiles handle how a Bigtable instance should handle requests from an application, such as routing to a specific cluster [4].

Unlike a MySQL cluster where data is always written to the primary, Bigtable data is written to the instance, which ensures that those changes are propagated to all the other clusters. Although

you can address specific clusters directly if needed, it shouldn't be necessary because Bigtable should route queries to the closest cluster and, therefore, be reliably fast. Instances are globally scoped, meaning that they remain addressable regardless of whether a particular zone is experiencing an outage [4].

2.3 Clusters

Clusters live under an instance of Bigtable and represent the service in a particular location. Each cluster has a unique name, a location (unique zone), and some performance settings, such as the number of nodes to run. Clusters themselves have an hourly computing cost, as well as a monthly storage cost to reflect the amount of data stored. Each cluster holds a copy of the data, and utilizing more clusters would increase the availability, albeit at higher running costs [4].

2.4 Nodes

Each cluster houses a number of nodes. Nodes are compute resources that are used to manage the instance data and are automatically managed behind the scenes by Google. The only option that can be configured by a user is the quantity that is allocated to a cluster. An advantage of the Bigtable system is that there will never be a bottleneck of "too many nodes," as has been known to happen with other systems such as HBase [6]. This structure allows the cluster to ensure that requests are spread evenly across all of its computer resources. If nodes themselves were addressable, the cluster wouldn't be able to move data around as freely, which could lead to a case where a single node held all the hot data, driving down performance during busy times [6].

2.5 Tablets

Tablets are a way of sharding chunks of data that live in an instance onto a particular node. They contain blocks of contiguous rows and are stored in Google's file system called Colossus, in SSTable format. The SSTable format "provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings" [2]. Tablets can be split, combined, and moved around to other nodes to keep access to data spread evenly across the available capacity. As with nodes, tablets cannot be individually addressed and are stored independently of nodes. If a node fails, the data inside the tablet is not affected [2].

3 DISTRIBUTED FEATURES

Bigtable was designed to handle and store the vast amounts of data a company such as Google needs to query. From a handful of machines to thousands, petabytes of data can be reliably scaled in Bigtable. In addition, the system was designed to hit important goals such as: "wide applicability, scalability, high performance, and high availability" [5].

3.1 High Availability

3.1.1 Distribution. To distribute the load of a Cloud Bigtable instance, data is split into 1 or more nodes. In the back end, this data is separated into multiple tablets, which are stored on disks separate from nodes, but located in the same zones. Each tablet is linked with a single node, and that node is responsible for keeping track of their tablets, handling the reads and writes for the tablets, and performing the maintenance tasks for the tablets [3]. This structure for bigtable instances allows the distribution of compute resources for handling many requests. If an instance cannot keep up with the current load, then latency suffers and requests may not be serviced. To increase the available compute for servicing requests, more nodes need to be added to the instance.

3.1.2 Replication. In the commercial offering of Bigtable, a cloud instance can have up to 4 replicated clusters located in Google cloud zones. When these clusters are created, Bigtable will automatically transfer and replicate data to each other cluster. This can be done even when adding new clusters to an existing instance. Bigtable is also not limited to having a single primary cluster to which reads and writes can be done. Every cluster is treated as a primary cluster with the added flexibility of routing specific requests to specific clusters if needed. This allows services to either use the Bigtable replication as a means to improve availability, by having multiple clusters servicing requests, or having an almost real time backup available in the event of cluster loss [3].

3.2 Performance

3.2.1 Speed (latency). One of the big draws of using Bigtable is its performance. By sacrificing the ability to write complex queries or operate atomically on more than a single row means that reading a single row is incredibly fast (typically below 10 ms, even with thousands of writes per second). Each table has only one index, the row key. There are no secondary indices, and the data is stored in the row key in ascending order. All operations are atomic at the row level. Hence every read operation performed only needs to match row keys that are stored in ascending order [2]. Though some in-memory storage systems are capable of this, few can maintain this level of speed without sacrificing durability or concurrency. For example, the performance of random reads from memory increases by almost a factor of 300 as the number of tablet servers increases by a factor of 500 [5].

The system is able to keep latency low by automatically moving frequently accessed data around to nodes with more compute availability; thereby keeping request loads evenly distributed. The rebalancing operation is often very fast, because no data is actually moved around, only the pointers are adjusted for each node [2].

3.2.2 Throughput. Throughput on Bigtable is best in class for storage systems. Bigtable is able to scale more easily to a larger number of nodes and, as a result, can handle more overall throughput for a given instance [6] .The same aspects of data redistribution that help to keep latency low also helps keep throughput high. Bigtable has the option to use solid state drives in which random reads and writes are extremely fast, and many of them can happen concurrently. By combining the high performance of the low-level storage with the load balancing across tablets, Bigtable clusters as a whole can handle extraordinarily large levels of throughput, with measurements starting in the tens of thousands of reads or writes per second [3]. Further, adding more capacity to the cluster is as simple as adding more nodes. As mentioned previously, Bigtable accounts for empty and idle nodes and shifts tablets to them based on the amount of traffic they currently receive. At the end you

Event Name	RPCs	Total Duration (ms)
opencensus.BigTable	1	19.097
WriteRows	1	12.355
Bigtable Table.put	3	12.185
Operations.google.bigtable.v2.Bigtable.MutateRow	3	12.047
Sent.google.bigtable.v2.Bigtable.MutateRow	3	11.91
ReadRows	1	6.579
Operations.google.bigtable.v2.Bigtable.ReadRows	2	6.286
Sent.google.bigtable.v2.Bigtable.ReadRows	2	6.169
Bigtable Table.scan	1	3.528
Bigtable Table.get	1	2.959

Table 1: Details of low level Remote Procedure calls

have a larger cluster with traffic evenly balanced across each node, improving your overall throughput.

3.3 Consistency and Fault Tolerance

3.3.1 Consistency. Bigtable uses an eventually consistent consistency model. Changes that are written to one cluster can eventually be read from another, but only after the change is propagated first. Google describes this delay as being a few seconds, but can be affected by the load on clusters. If a cluster is overloaded, then change replication can take longer to complete [2].

Google also offers a consistency guarantee using the client centric model 'read-your-writes'. This makes sure that when an application is reading data from a cluster it will never read data that is older than its most recent writes. To configure this consistency, the application must use an app profile using single-cluster routing and must route to the same cluster. There is also the possibility to use strong consistency, but you cannot use the instance's other clusters unless for failover protection.

3.3.2 Fault Tolerance. If a particular cluster becomes unresponsive or goes down, the instance can be configured to reroute traffic to another cluster. This can be done in an automatic or manual fashion and is configured in the app profile of an application. For data backups, cluster replication is required, as bigtable does not provide out of the box data backups [2].

4 RESULT

Although Bigtable surfaces several helpful server-side metrics using Trace, apps can implement client-side tracing, instrumenting, and app-defined metrics. Client-side tracing gives you a window into the round-trip latency of calls made to your Bigtable endpoint. A simple java app, provided by Google [1], running on a VM does the following:

(1) Creates a table in Bigtable
(2) Write a small set of rows
(3) Read a single row
(4) Performs a table scan for those rows
(5) Deletes the table

The sample program performs 10,000 transaction sets: three writes and a range read. The exporter is configured to record one trace sample for every 1,000 transaction sets. As a result, 10 or 11 traces are captured over the run of the program. Below is the summary of one of the trace which shows tracing scopes for discrete API calls instrumented in the client library and operation-level call latencies. Each of the series of write and read rows are encapsulated by the lower-level, user-defined WriteRows and ReadRows tracing spans.

Table 1. Traces details of low level Remote Procedure calls taking place while performing a set of read/write operations on a Bigtable Instance opencensus. Bigtable is the trace label that displays the total time taken for the entire trace of 1000 sets of read/write operation.

REFERENCES

[1] [n.d.]. Data Source. https://github.com/GoogleCloudPlatform/community/tree/master/tutorials/bigtable-oc. Accessed: 2021-03-16.
[2] [n.d.]. Google Cloud - Application Profiles, Instances, Clusters, Nodes, Overview of Replication. https://cloud.google.com/bigtable/docs. Accessed: 2021-02-18.
[3] [n.d.]. Google Cloud - Overview, Architecture. https://cloud.google.com/bigtable/docs/overview. Accessed: 2021-03-16.
[4] [n.d.]. Pricing. https://cloud.google.com/bigtable/pricing. Accessed: 2021-01-30.
[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
[6] John J Geewax. 2018. Google Cloud Platform in Action. (2018).