Analysis of Distributed Properties of Google Bigtable

Akshay Dhabale (40163636)
Concordia University
Montreal, Canada
akshaydhabale@gmail.com

Kshitij Yerande (40194579)
Concordia University
Montreal, Canada
kshitij.yerande@mail.concordia.ca

Mrinal Rai (40193024)
Concordia University
Montreal, Canada
mrynalrai@gmail.com

Yogesh Yadav (40202192)
Concordia University
Montreal, Canada
yogeshoyadav08@gmail.com

***Abstract*—Data has evolved exponentially in recent years and it has become difficult to manage the size of data using traditional databases due to storage limitations and concerns regarding aspects such as scalability, availability, reliability, and fault tolerance. Distributed databases have been developed to address the concerns of managing huge data sets. Google Bigtable is one such distributed database to manage data at a very large scale. In this report, we discuss the distributed aspects of Google Bigtable by analyzing the Amazon review data set followed by a discussion on architecture, data model, and implementation of Google Bigtable.**

***Index Terms*—Google Big Table, Distributed Storage Systems, Distributed Database, Chubby, Tablets**

I. Introduction

In this project, we use amazon review data set [2] to study the distributed aspects of Google Bigtable such as auto-scaling, replication, automatic failover, and observing CPU Utilization, read/write latency, and replication latency while loading data. Additional study will be performed on the execution time of queries with varying complexities.

Section 2 describes the data model of Bigtable and gives a brief overview of API in more detail. Section 3 describes the building blocks of Bigtable: Google File System, Chubby Lock Service, and Map Reduce. Section 4 discusses the implementation and underlying infrastructure of Bigtable while Section 5 mentions about the refinements done in Bigtable to improve performance.

II. Data Model and API

Bigtable stores data in massively scaled tables, each of which is sorted key/value map. The table is composed of rows, each of which describes a single entity, and columns that contain individual values for each row. Each row-column intersection can contain multiple cells with the unique time for that row and column data. Bigtable tables are sparse if a column is empty for row key, it does not take up any space. In our study of the Amazon Review Data set, we came up review ID as a row key for our data model which is unique for every record in the data set. Various other fields from data sets are columns that provide information related to each review.

A. Rows

The row keys in a table are arbitrary strings (currently up to 64KB in size). Row keys are the unique identifier for a row of data stored in Bigtable. Row keys identify where the data is written in Bigtable. As Big Table does not support the join operation, it does not contain any foreign keys. Bigtable maintains data in lexicographic order by row key. Every read or write operation under a single row key is atomic regardless of the number of different columns being read or written in the row. Bigtable writes data to multiple nodes, row keys are useful for determining write operations on various nodes. Row keys are evenly distributed across the nodes. Row keys play an important role while developing tables in Bigtable. Some of the design considerations while designing row keys are avoiding linearly incrementing keys and low cardinality attributes. Avoid using time series values as row keys this will lead to writing adjacent row keys to the same node and we will lose parallel processing capabilities of Bigtable. Instead, a better design approach would be to concatenate multiple attributes.

B. Column Families

Columns in the Bigtable are grouped together called Column Families. Access control and disk/memory accounting are performed at the column-family level. A column family must be created before data can be stored under any column in the column family. The general design approach is to have a smaller number of column families typically few to hundreds. However, the number of columns is not limited. Each column in Bigtable is identified using a combination of column family and column qualifier which is always unique in a column family. Column family name must be printable, but qualifiers may be arbitrary strings. Data stored under column family is usually of the same type.

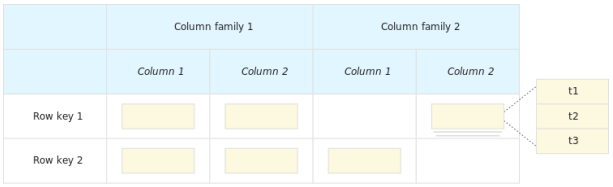

Fig. 1. Diagram showing rows and columns in Bigtable [5]

C. Timestamp

Each cell in Bigtable can contain multiple versions of data with the timestamp. Bigtable timestamps are 64-bit integers. Timestamps can be assigned by Bigtable automatically or client can use their timestamps specified by the application. In case of timestamps being generated from the application, it should produce a unique timestamp to avoid the collision. In Bigtable data is stored in decreasing timestamp order keeping recent data first before older data. This also helps to retrieve the recent data quickly. A table is configured with per-column-family settings for garbage collection of old versions. A column family can be defined to keep only the latest n versions or to keep only the versions written for some time t.

D. API

The Bigtable APIs provide functions for creating, deleting tables and column families. It also provides functions for changing cluster, table, and column-family metadata such as access control. [1] Bigtable also provides several other features which allow clients to manipulate data in a more complex way. Bigtable supports single-row transactions which are used to perform atomic read-modify-write operations. Bigtable also supports client-supplied scripts which can be executed in the server space. These scripts are written in a language called Sawzall. [6]

III. Architecture

A simpler diagram of Google BigTable architecture is attached in Fig. 2.

As shown in the flow diagram, all the client requests go through a front-end server before subsequently reaching a Bigtable node. They are organized into a BigTable cluster which belongs to a Bigtable instance. Each node (also referred to as ”tablet servers”) in the Bigtable cluster handles a subset of requests sent to the cluster. The nodes are added to increase the number of simultaneous requests in a cluster to handle maximum throughput.

The Bigtable table is sharded into blocks of contiguous rows called tablets. It helps in balancing the workload of queries. Tablets are stored on colossus GFS, in SSTable format. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. In addition to the SSTable files, all writes are stored in Colossus’s shared log as soon as they are acknowledged by Bigtable, providing increased durability. The data is never stored in nodes themselves. Each node has pointers to a set of tablets stored on colossus. Recovery from the failure of the node is very fast. Whenever there is a failure in Bigtable, no data is lost. [5]

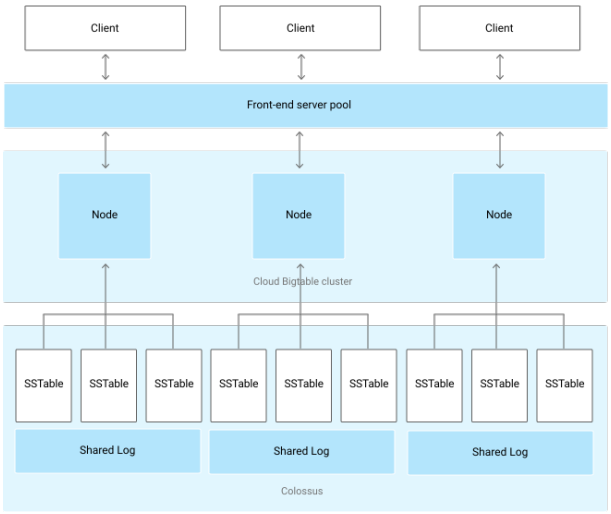

Fig. 2. Google Bigtable overall architecture [5]

A. Google File System

Google File System is essentially a distributed File storage. In any given cluster of GFS, there can be thousands of commodity servers. And these servers provide an interface for n number of clients to either read a file or write a file.

There were multiple design considerations made while designing the DFS architecture. Firstly, is the commodity hardware. Instead of buying an expensive server, google (as a young company) chose to buy off-the-shelf commodity hardware because of its low price. And secondly, using a lot of these servers they can scale horizontally by using the right software layer created on top of it. The third important design consideration was that GFS is optimized for two types of file operations. One, the writes to any files are generally append-only; there are hardly or no random writes in the file. Two, the reads are sequential.

In DFS, a single file is not stored on a single server. It is sub-divided into multiple chunks and these chunks can be shared across multiple machines called chunk servers. The servers are not storing an entire file but chunks of a particular file. And since these files are stored on commodity hardware (which can go down any time), GFS ensures that each chunk of your file has at least three replicas (number of replicas are configurable) across three different servers so that even if one server goes down you still have other two replicas to work with. GFS provides a separate component, GFS master server, which has all the metadata, which contains the names of the files which are there in that particular cluster. It is important to note the GFS client does not read the data from the GFS master. It only reads metadata about the file. The actual reading/writing of the file is directly done between the client and the GFS chunk server. [8][9]

B. Map Reduce

In the previous section, we discussed the overview of the Google File System. It is used to store a huge amount of data which is difficult to achieve with vertical scaling. And because now these files are distributed across machines, it also helps in doing batch processing much faster than a traditional server, and that batch processing system is called Map Reduce.

The programming model where instead of taking out all the data from multiple servers, the code is pushed onto multiple servers, and those servers are requested to process or run that code is called MapReduce. This is how multiple large files with terabytes of data can be processed in a very short amount of time because there are hundreds or thousands of servers working simultaneously or parallelly on individual parts of the file. The name MapReduce comes from two parts where the first one is the map function. The map function is the processing function that the client passes to the servers. The second part is the reduced function which is the aggregation of all the results from each of the individual servers. [10]

C. Chubby

Chubby is a highly-available and persistent distributed lock service that is used by Bigtable. It consists of five active replicas, among which one of the replicas is elected as the Master and listens to active requests. The service is live when a majority of the replicas are running and can communicate with each other. In situations where Chubby is not available for a long period, Bigtable also becomes unavailable. Chubby is utilized by Google Bigtable for a variety of functions:

- ensuring that there is at most one master active at any given time
- discovering tablet servers and finalizing tablet server deaths
- storing schema information of Bigtable
- storing access to control lists

Chubby provides fault tolerance by using the Paxos algorithm. It also provides a namespace that consists of directories and small files. Each directory or file can be used as a lock, and reads and writes to a file are atomic. [1]

IV. Fundamentals of Bigtable Implementation

The Bigtable internal system implementation has 3 components: a client request manager, master server, and tablet servers. The client manager(library) is used to handle requests from the client. The master server is used to manage the addition and deletion of tablet units to the tablet server, manage load balancing of the tablet server, and garbage collection of files in GFS. The tablet server is responsible for handling read/write requests from the client. In Bigtable, client requests are directly handled by the tablet server for reading and writing operations. Most of the time, clients never communicate with the master and thus master is light weighted.

A BigTable cluster consist of multiple tablet server. Each tablet server consists of set of tablets. These tablets hold data associated with similar rowkey. Based on rowkey, data is distributed across tablet nodes, each node containing similar data for table scan. Initially, each tablet node includes only one table but as table grows in size, it automatically splits into multiple tablets approximately 100-200MB in size.

A. Tablet Location

A three-level hierarchy to store tablet location information is shown in Fig. 3 The Chubby contains the location of the

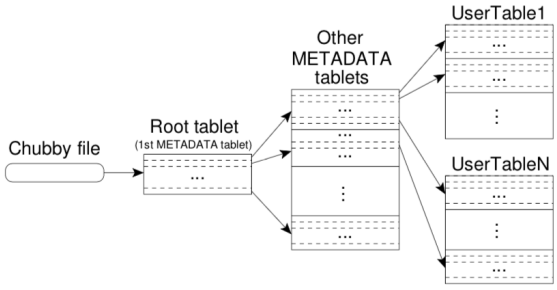

Fig. 3. Tablet Location Hierarchy

root tablet(First level). Root tablet contains location details of METADATA tablets. Each METADATA tablet contains locations of user-defined table. The hierarchy is maintained by not allowing splitting of root tablet.

On client request, the client manager caches user-defined tablet location. In case the cache is empty or incorrect or stale, the locations algorithm requires multiple round trips across location hierarchy levels to correct the data in the cache.

B. Tablet Assignment

In Tablet assignment process, each tablet is assigned to a live tablet server. As mentioned, the master keeps track of all live tablet servers, current tablets associated with them, also including which tablets are still not assigned to any tablet. When is a vacancy in the tablet server, the master assigns a tablet from the unassigned list by sending a tablet load request to the server.

The internal system uses chubby to keep track of tablet servers. When a tablet server starts, it creates and acquires an exclusive lock on the uniquely named file in the chubby directory. The tablet server stops serving if it loses its exclusive lock due to a network error. Chubby has an in-build mechanism that allows each tablet server to check its lock file. If the tablet server lost its access, it can reacquire the lock if the file still exists inside chubby. If a file is not found, the tablet server kills itself and frees all assigned tablets. Master monitors each tablet server and as soon as the tablet server terminates, it gains access to free tablets to be reassigned to another tablet server.

The Master asks each tablet periodically for the status of its lock. If the tablet server has lost the access to lock or the master is not able to reach to tablet server after several attempts, the master gains exclusive access to the lock, and then if chubby is live and the master is either dead or unable to access the lock. Master then terminates the server by deleting

the file inside chubby. The master then includes all free tablets into an unassigned set of tablets. To ensure the Bigtable cluster is not vulnerable to network issues, the master itself kills if it loses its lock on chubby.

As mentioned in Google BigTable's original paper[1]-"Cluster Management System restarts Master. After Master restarts, it executes the following steps (1) The master grabs a unique master lock in Chubby, which prevents concurrent master instantiations. (2) The master scans the servers directory in Chubby to find the live servers. (3) The master communicates with every live tablet server to discover what tablets are already assigned to each server. (4) The master scans the METADATA table to learn the set of tablets. Whenever this scan encounters a tablet that is not already assigned, the master adds the tablet to the set of unassigned tablets, which makes the tablet eligible for tablet assignment".

Master scans root tablet before step 4 so that location of METADATA tables are known.

As mentioned, Each tablet is allocated to a single table. As the table grows or shrinks in size, tablet addition/deletion takes place. Master keeps track of all these changes.

C. Tablet Server

Each Tablet state information is stored in GFS. There is a commit log to record all tablet updates. In addition, the most recent updates are stored in memory called a memtable. To recover a tablet, the tablet server reads GFS commit log details into memory and reconstructs the commit changes carried out by the tablet. The tablet server is responsible for reading and writing requests from the client. For each read request, it checks for well-formedness and authorization. It reads data using memtable and SSTable(GFS Logs). Similarly, for every writes operation after checking authorization and well-formedness, it writes into a commit log file of the tablet. Incoming read and write operations can continue to execute independently of merging/splitting of tablets

D. Compaction

As mentioned in tablet server, most recent updates(commits(read/write) operations) are stored in memtable and older updates into SSTables(GFS). Over a period of read/write operations, the size of memtable reaches a threshold. Once it reaches a threshold, memtable freezes, and a new memtable is created by merging a few memtables. Frozen memtable is converted into an SSTable(older updates). This phenomenon is known as Compaction. Minor compactions improve memory usage of tablet server and reduce commit log data recovery if the server dies. To ensure not every minor compaction results in SSTables, periodically we merge SSTables and memtables into new SSTables. This is known as merging compaction. Merging compaction that rewrites all SSTables into exactly one new SSTables is called Major Compaction.

E. Case Study Google BigTable: Amazon Reviews Data study and implementation on Google Ecosystem

Amazon Customer Reviews is one of Amazon's iconic products. In a period of over two decades since the first review in 1995, millions of Amazon customers have contributed over a hundred million reviews to express opinions and describe their experiences regarding products on the Amazon.com website. This makes Amazon Customer Reviews a rich source of information for academic researchers in the fields of Natural Language Processing (NLP), Information Retrieval (IR), Information Processing(IP), and Machine Learning (ML), amongst others.The data set represents customer evaluations and opinions, variation in the perception of a product across geographical regions, and promotional intent or bias in reviews.

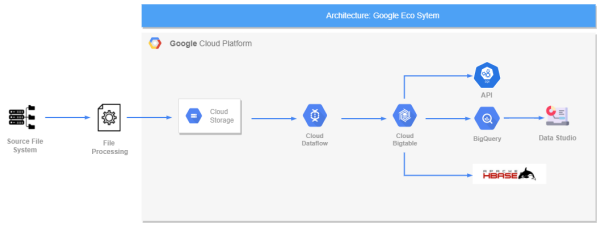

Fig. 4. Data Flow Diagram

The objective of the case study is to understand, study and implement the distributed aspects of Google BigTable such as Auto-scaling, High Availability, Automatic Fail-over, High Read-Write Throughput etc.on amazon reviews data set.

As seen in Fig. 4. In source system, files were available in TSV format. In file processing phase, we carried out data cleaning to remove unwanted column attributes, null values, special characters, reordering of columns and converting to CSV format. The source files were then loaded into Google Cloud Storage using Google Cloud Console. In Google BigTable, an instance is created with required system requirements in a particular region and zone. Auto Scaling could also be enabled in an instance specifying target number of nodes to scale and CPU utilisation as per the system requirement. Additionally, a replication cluster could also be set up for high availability of data across a difference region/zone. Once a BigTable instance is set up, we could create a table using CBT tools/HBase API on Google Cloud Shell. Column Families mentioned in 2.A are specified for the table and Row-key as mentioned in 2.A is identified and added as a first column in cleaning phase in the source data. Google Data Flow Job is executed which loaded data from the Google Cloud Storage bucket into the Google BigTable. Once data is loaded into the table, querying data could be carried out in various ways such as using CBT tools, HBASE, external API calls and creating external permanent tables in Google BigQuery. Extensive Read Operations on BigTable using BigQuery could be carried out to gain better understanding of system performance and table scan. This insights in turn could help in improving

performance of the system by taking data driven decision in incorporating column families and row-key on the table.

V. Refinements

Certain refinements were required in Bigtable to achieve high performance, availability, and reliability required by users.

A. Locality Groups

Column families can be grouped together into a locality group which is stored in a separate SSTable on each tablet. These are similar to indexes on a normal relational database. Segregating column families which are frequently read can be grouped together to increase read throughput. For example, metadata can be stored in one locality group and contents can be stored in another locality group, therefore, an application requiring to access metadata needs not to scan the whole contents, therefore, improving read efficiency.

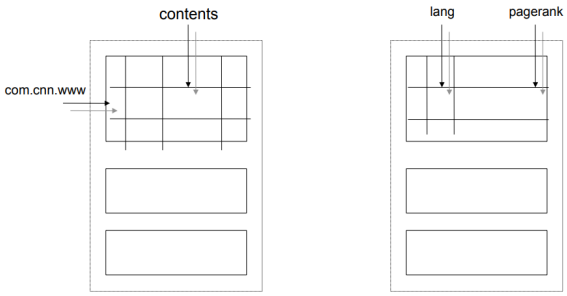

Fig. 5. Locality groups in Bigtable

Further locality groups can be stored in memory. SSTables for in-memory locality groups is loaded lazily into the memory of the tablet server. Once loaded, column families that belong to such locality groups can be read without accessing the disk. This is useful for accessing small data like METADATA.

B. Caching

Two levels of caching are implemented to improve read performance. Scan Level cache implemented at a high level which caches key-value pair at SSTable interface which is returned to Table Server code. These are useful in case the application is reading the same data frequently. The low-level cache also called a Block Cache which caches SSTable blocks in Google File System. This is useful for applications that tend to read data that is close to the data they recently read.

C. Bloom Filters

There are multiple read operations happening on SSTables and in case these tables are not stored in memory then there will be multiple disk access due to read operation. A better way is to apply a bloom filter on SSTables in a particular locality group. A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set by applying hash functions on the data item [3].

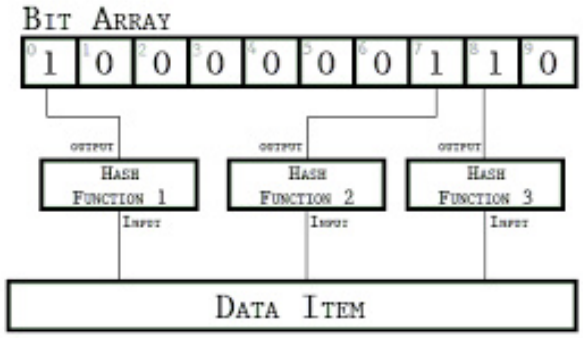

Fig. 6. Bloom Filter

In Bigtable a Bloom filter allows us to check whether a particular data item is present in the SSTable.For certain applications, a small amount of tablet server memory used for storing Bloom filters drastically reduces the number of disk seeks required for read operations.

D. Faster Tablet Recovery

While moving tablets between tablet servers there is a minor compaction performed on the commit logs. This reduces the amount of uncompacted state of the commit logs. After the tablet server stops serving the tablet there is another minor compaction performed on the remaining uncompacted state of the tablet server before it unloads the tablet server. After this second minor compaction is complete, the tablet can be loaded on another tablet server without requiring any recovery of log entries.

E. Immutability

SSTables are immutable in nature to avoid concurrency issues. For example, the only mutable data structure is the memtable which is accessed during reading and writing operations. Thus no synchronization is required while accessing SSTables. To reduce contention between reading and write operations, the copy on write approach is implemented resulting in parallel read and write operations.

VI. Results

Following observations were made regarding features of big table while processing the amazon reviews data set.

A. Auto Scaling

Scaling a cluster is a process of adding or removing nodes from cluster with varying workloads. Bigtable offers automatic scaling as well as manual scaling of nodes through google console [4]. Following steps were carried out in implementing auto-scaling.

1) Create a BigTable instance with auto-scaling capability. Target nodes could be defaulted to 3 or a greater number depending on the system requirement. Target CPU Utilisation's added as 10% or above, meaning if CPU utilisation at node 1 is higher than 10% or the

targeted value, a new node will be spin up for balancing the load.

2) Once data load in bigtable is completed, we could view statistics of auto-scaling and number of nodes used in the Monitoring view in the Google Bigtable Console.

Fig. 7. Auto scaling

We can observe that as the CPU utilization crosses the 10% threshold the number of nodes increases to 3 from 1. Similar for replicated cluster as well and once the utilisation drops down for replicated cluster the number of nodes decreases to 1. Therefore the number of nodes changes with varying workloads.

B. Automatic Fail over

If a Cloud Bigtable cluster becomes unresponsive, replication makes it possible for incoming traffic to fail over to another cluster in the same instance. Fail overs can be either manual or automatic, depending on the app profile an application is using and how the app profile is configured [10].

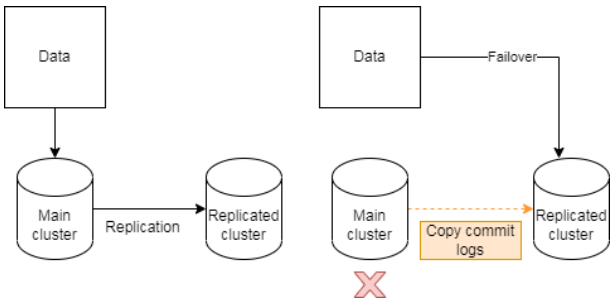

Fig. 8. Fail over

We have implemented automatic fail over by creating two clusters where one is acting as main cluster (dsdtestcsvimport-c1) and other as replicated cluster (dsdtestcsvimport- c2). Application profile was configured to fail over to nearest cluster. While data loading was in progress in main cluster we deleted the main cluster. We observed that the data loading failed over to the replicated cluster.

It is been observed that during fail over CPU utilisation of main cluster begins to drop while the replicated cluster rises.

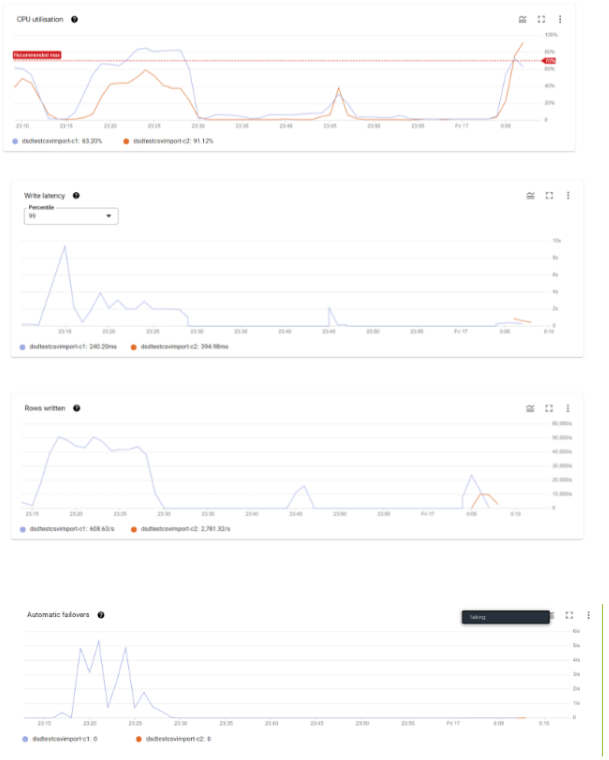

Fig. 9. Failover Monitoring

The same pattern is observed in write latency and number of rows written which concludes that the fail over has started at the particular moment. The fail over graph shows the time at which fail over happened. There is a small orange line at the bottom right of the graph which shows that fail over has happened from between main and replicated cluster.

C. Read Write Throughput

Cloud Bigtable as the storage engine provides low latency and high throughput for data processing and analytics. In the implementation, amazon reviews data of size approx 2.5 GB and records count 30M were loaded over a Google Bigtable instance with the system configuration of single node across region US-Central. We observed below Read/Write Latency and Read/Write throughput for our data load.

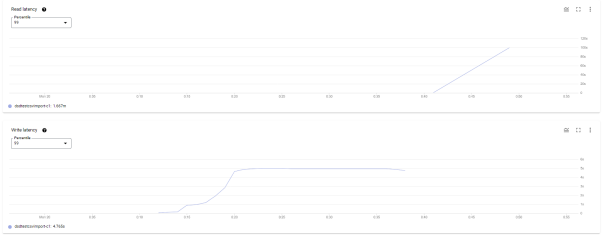

Fig. 10. Read Write Latency

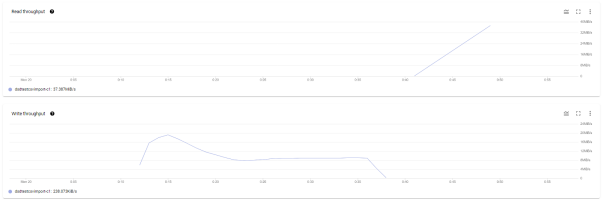

Fig. 11. Read Write Throughput

As per above fig. 10 and Fig. 11, In our data loading to bigtable, we observe a stable write latency around 5 seconds and write throughput raging from 8 MiB/s to 20 MiB/s and gradually coming down to stable 8 MiB/s over the load.

D. Query Statistics

In BigTable, we performed extensive query statistics to analyse the performance of Bigtable read scans. We observed below query performance statistics for diverse query table scans in Google BigTable.

Query Statistics			
Table Scan/Read Query	BigQuery	CBT	Filter-API
Approx Response Time on 2.5 GB Data			
	Response Time(sec)	Response Time(sec)	Response Time(sec)
Entire Table Scan	300	>900	5000
Count Table Records	110	525	3600
Scan 5M records	240	>900	3600
Distinct Product ID Scan	135	n/a	240
Aggregate Product ID Scan	189	n/a	n/a
Filter Product ID Scan	215	n/a	320
Time Range Scan	225	n/a	600

VII. Conclusion

We have studied distributed aspects like auto-scaling, automatic failover, read-write throughput, and query performance statistics of Bigtable by using amazon reviews data set. Bigtable is designed for handling very large data, however, we have tried to study the distributed aspects using roughly 1.5 - 2.5 GB of data due to cost limitations. Bigtable scales by adding or removing nodes automatically to cluster if the CPU utilization goes beyond the threshold set by the user while creating an instance. Data is replicated across clusters if replication is enabled by the user and automatic failover happens in case a node or a cluster is offline. users can specify the location of the replicated cluster and can also configure the failover cluster. Big Table provides table schema designing using row key and column families to help in improving distributed computing and system performance. Bigtable provides a user-friendly user interface to manage various aspects of Bigtable. It provides a monitoring interface that is used to observe the performance of clusters over time. Big table data can be sourced into BigQuery to run high-performance SQL queries. There is a client API that can also be used to query Bigtable. While studying Bigtable we have come across various research papers, online resources, and presentations that have given us a detailed overview of the distributed NoSQL system.

References

[1] Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A distributed storage system for structured data." ACM Transactions on Computer Systems (TOCS),2008

[2] https://www.kaggle.com/cynthiarempel/amazon-us-customer-reviews-dataset

[3] https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/

[4] https://cloud.google.com/bigtable/docs/autoscaling

[5] https://cloud.google.com/bigtable/docs/overview

[6] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN,S. Interpreting the data: Parallel analysis with Sawzall. Scientific Programming Journal 13, 4 (2005)

[7] Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003.

[8] Vadgama, Deepak. "Google File System - Paper that inspired Hadoop" YouTube, uploaded by Defog Tech, 28 April 2019, https://www.youtube.com/watch?v=eRgFNW4QFDc&t=24s.

[9] Vadgama, Deepak. "Map Reduce Paper - Distributed data processing" YouTube, uploaded by Defog Tech, 4 May 2019, https://www.youtube.com/watch?v=MAJ0aW5g17c.

[10] https://cloud.google.com/bigtable/docs/managing-failovers.